O'REILLY®

TURING 图灵程序设计丛书

流畅的Python

Fluent Python

PSF研究员、知名PyCon演讲者心血之作

全面深入，对Python语言关键特性剖析到位

[巴西] Luciano Ramalho 著

安道 吴珂 译





版权信息

书名：流畅的Python

作者：[巴西] Luciano Ramalho

译者：安道　吴珂

ISBN：978-7-115-45415-7

版权声明

O'Reilly Media, Inc. 介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。**”**

——*Wired*

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。**”**

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。**”**

——*CRN*

“一本 **O'Reilly** 的书就代表一个有用、有前途、需要学习的主题。**”**

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 **Yogi Berra** 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，**Tim** 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。**”**

——*Linux Journal*

致 **Marta**，用我全心全意的爱。

前言

> 要不这样吧，如果编程语言里有个地方你弄不明白，而正好又有个人用了这个功能，那就开枪把他打死。这比学习新特性要容易些，然后过不了多久，那些活下来的程序员就会开始用 0.9.6 版的 Python，而且他们只需要使用这个版本中易于理解的那一小部分就好了（眨眼）。[1]

—— Tim Peters
传奇的核心开发者，“Python 之禅”作者

> [1]给 comp.lang.python Usenet 小组的留言，2002 年 12 月 23 日，“Acrimony in c.l.p”（https://mail.python.org/pipermail/python-list/2002-December/147293.html）。

Python 官方教程（https://docs.python.org/3/tutorial/）的开头是这样写的：“Python 是一门既容易上手又强大的编程语言。”这句话本身并无大碍，但需要注意的是，正因为它既好学又好用，所以很多 Python 程序员只用到了其强大功能的一小部分。

只需要几个小时，经验丰富的程序员就能学会用 Python 写出实用的程序。然而随着这最初高产的几个小时变成数周甚至数月，在那些先入为主的编程语言的影响下，开发者们会慢慢地写出带着“口音”的 Python 代码。即便 Python 是你的初恋，也难逃此命运。因为在学校里，抑或是那些入门书上，教授者往往会有意避免只跟语言本身相关的特性。

另外，向那些已在其他语言领域里有了丰富经验的程序员介绍 Python 的时候，我还发现了一个问题：人们总是倾向于寻求自己熟悉的东西。受到其他语言的影响，你大概能猜到 Python 会支持正则表达式，然后就会去查阅文档。但是如果你从来没见过元组拆包（tuple unpacking），也没听过描述符（descriptor）这个概念，那么估计你也不会特地去搜索它们，然后就永远失去了使用这些 Python 独有的特性的机会。这也是本书试图解决的一个问题。

这本书并不是一本完备的 Python 使用手册，而是会强调 Python 作为编程语言独有的特性，这些特性或者是只有 Python 才具备的，或者是在其

他大众语言里很少见的。Python 语言核心以及它的一些库会是本书的重点。尽管 Python 的包索引现在已经有 6 万多个库了，而且其中很多都异常实用，但是我几乎不会提到 Python 标准库以外的包。

目标读者

本书的目标读者是那些正在使用 Python，又想熟悉 Python 3 的程序员。如果你懂 Python 2，但是想迁移到 Python 3.4 或者更新的版本，也没问题。在写这本书的时候，大多数专业 Python 程序员用的还是 Python 2，因此如果书中出现来自 Python 3 的特性，读者可能会感到陌生，我也会特别地做出解释。

然而，本书的主要目的是为了充分地展现 Python 3.4 的魅力，因此我不会一字一句地说明如何让本书的代码在旧版本里正常运行。本书中的大多数例子稍做修改（甚至不用修改）就可以在 Python 2.7 里面跑起来，但是有些例子，如果追求向下兼容，就会需要大量的重写。

话虽如此，我还是认为，即便你无法从 Python 2.7 里脱身，这本书也会对你很有帮助，因为 Pyhon 语言的核心概念是不会变的。Python 3 也不是一门全新的语言，大多数的改动花一下午大概就能适应，官方文档里“Python 3.0 的新特性”一节（https://docs.python.org/3.0/whatsnew/3.0.html）就是很好的切入点。固然，自 2009 年发布以来，Python 3.0 也在变化，但是这些变化比起 Python 3.0 和 Python 2.0 之间的区别，并没有那么重要。

如果你尚不清楚自己对 Python 的熟悉程度能否跟得上本书的内容，建议你回头看看 Python 的官方教程。注意，除非是跟 Python 3 的新特性有关，教程里的其他内容本书不会重复。

非目标读者

如果你才刚刚开始学 Python，本书的内容可能会显得有些“超纲”。比难懂更糟的是，如果在学习 Python 的过程中过早接触本书的内容，你可能会误以为所有的 Python 代码都应该利用特殊方法和元编程（metaprogramming）技巧。我们知道，不成熟的抽象和过早的优化一样，都会坏事。

本书的结构

如果你是本书的目标读者，那你应该可以从本书的任意一章开始阅读，但是如果按照我写作时的构思来的话，本书一共分为六个独立的部分，每个部分内的章节最好按照顺序来读。

在介绍让你自己实现某些功能的方法之前，我通常会先把现成可用的工具讲清楚。比如说第二部分的第 2 章覆盖了序列类型（sequence type），但是像 `collections.deque` 这种类可能就会一带而过。一直到第四部分，我们才会看看如何从抽象基类（abstract base class，ABC）中获利，抽象基类则被封装在 `collections.abc` 这个包里。如果想创建自己的 ABC，你可能得看到第四部分的最后一些内容才行，因为我一直觉得，如果没有熟练使用 ABC 的经验，贸然去实现一套自己的东西是不合适的。

这样做有几个好处。第一，知道有什么现成的工具可用，能避免重新发明轮子。毕竟我们使用现有集合类型（collection type）的概率要远大于自己动手写一套新的。第二，这样一来，在讨论如何写新类型之前，我们能够有更多的机会来了解这些现成类的高级用法。第三，比起从零开始构建一个 ABC，继承已有的 ABC 库应该会简单一些。最后，我认为在看过一些实际的案例之后，理解抽象会更轻松。

当然，这样也会带来一些不便之处，比如书里的向前引用就会分散在各个不同的章节里面。但是经过上述这番梳理，我想这一点不便之处也是可以容忍的。

下面是本书每一部分的主题。

第一部分

　　第一部分只有单独的一章，讲解的是 Python 的数据模型（data model），以及如何为了保证行为一致性而使用特殊方法（比如 `__repr__`），毕竟 Python 的一致性是出了名的。其实整本书几乎都是在讲解 Python 的数据模型，第 1 章算是一个概览。

第二部分

第二部分包含了各种集合类型：序列（sequence）、映射（mapping）和集合（set），另外还提及了字符串（`str`）和字节序列（`bytes`）的区分。说起来，最后这一点也是让亲者（Python 3 用户）快，仇者（Python 2 用户）痛的一个关键，因为这个区分致使 Python 2 代码迁移到 Python 3 的难度陡增。第二部分的目标是帮助读者回忆起 Python 内置的类库，顺带解释这些类库的一些不太直观的地方。具体的例子有 Python 3 如何在我们观察不到的地方对 `dict` 的键重新排序，或者是排序有区域（locale）依赖的字符串时的注意事项。为了达到本部分的目标，有些地方的讲解会比较大而全，像序列类型和映射类型的变种就是这样；有时则会写得很深入，比方说我会对 `dict` 和 `set` 底层的散列表进行深层次的讨论。

第三部分

如何把函数作为一等对象（first-order object）来使用。第三部分首先会解释前面这句话是什么意思，然后话题延伸到这个概念对那些被广泛使用的设计模型的影响，最后读者会看到如何利用闭包（closure）的概念来实现函数装饰器（function decorator）。这一部分的话题还包括 Python 的这些基本概念：可调用（callable）、函数属性（function attribute）、内省（introspection）、参数注解（parameter annotation）和 Python 3 里新出现的 `nonlocal` 声明。

第四部分

到了这里，书的重点转移到了类的构建上面。虽然在第二部分里的例子里就有类声明（class declaration）的出现，但是第四部分会呈现更多的类。和任何面向对象语言一样，Python 还有些自己的特性，这些特性可能并不会出现在你我学习基于类的编程的语言中。这一部分的章节解释了引用（reference）的原理、“可变性”的概念、实例的生命周期、如何构建自定义的集合类型和 ABC、多重继承该怎么理顺、什么时候应该使用操作符重载及其方法。

第五部分

Python 中有些结构和库不再满足于诸如条件判断、循环和子程序（subroutine）之类的顺序控制流程，第五部分的笔墨会集中在这些构造和库上。我们会从生成器（generator）起步，然后话题会转移到上下文管理器（context manager）和协程（coroutine），其中会涵盖新增的功能

强大但又不容易理解的 `yield from` 语法。这一部分以并发性和面向事件的 I/O 来结尾，其中跟并发性相关的是 `collections.futures` 这个很新的包，它借助 `futures` 包把线程和进程的概念给封装了起来；而跟面向事件 I/O 相关的则是 `asyncio`，它的背后是基于协程和 `yield from` 的 `futures` 包。

第六部分

　　第六部分的开头会讲到如何动态创建带属性的类，用以处理诸如 JSON 这类半结构化的数据。然后会从大家已经熟悉的特性（property）机制入手，用描述符从底层来解释 Python 对象属性的存取。同时，函数、方法和描述符的关系也会被梳理一遍。第六部分会从头至尾地实现一个字段验证器，在这个过程中我们会遇到一些微妙的问题，然后在最后一章中就自然引出像类装饰器（class decorator）和元类（metaclass）这些高级的概念。

以实践为基础

一般情况下，我们会用 Python 的交互式控制台来探索各种库和语言本身。有些读者可能对静态的需要编译的语言更熟悉，但是这些语言可能不会提供 REPL（read-eval-print loop，读取、求值、输出的循环）。在这里我想强调一下 Python 交互式控制台，也就是 REPL，作为一个学习工具的重要性。

doctest（https://docs.python.org/3/library/doctest.html）是 Python 的一个标准库，做测试用的。这个库通过模拟控制台对话来检验表达式求值是否正确，而本书中几乎所有代码的测试，包括那些在控制台里的输出，都是通过这个库来进行的。doctest 看起来就像是 Python 交互式控制台的剧本，你甚至都不需要了解它背后的运行机制就可以直接用它来试验书里的例子。

我有时为了事先说明一段代码的目的，会在展示代码之前先摆出相应的 doctest 文本。这是因为我认为，在考虑如何实现一个功能之前，先严格地列出这个功能能做什么，这能帮助我们在编程时把精力花在该花的地方。测试驱动开发（TDD）的精髓就是先写测试，我后来发现这种精神在教学中也是大有益处的。如果你对 doctest 还不熟悉，花点时间阅读它的文档（https://docs.python.org/3/library/doctest.html）。结合本书的源码（https://github.com/fluentpython/example-code），你可以在操作系统的控制台里键入 `python3 -m doctest example_script.py` 来验证书中几乎所有代码的正确性。

硬件

书中有一些简单的时间和基准测试，跑这些测试的时候我用的是写书时的两台笔记本电脑。一台是产于 2011 年的 MacBook Pro 13 英寸笔记本，配置是 2.7 GHz 的英特尔 Core i7 处理器、8GB 的内存和机械硬盘；另一台是产于 2014 年的 MacBook Air 13 英寸笔记本，配置是 1.4 GHz 的英特尔 Core i5 处理器、4GB 内存和一个固态硬盘。MacBook Air 的处理器虽然慢一些，内存也没有另一台多，但是它的内存快一些（1600 MHz，MacBook Pro 13 英寸则是 1333 MHz），另外它的硬盘也更快，因此在日常使用中我并没有感觉到两台笔记本有速度上的差异。

杂谈：个人的一点看法

从 1998 年起，我一直在使用 Python，也做 Python 教学，另外还一直在为它辩护。我一直都很享受这个过程，尤其是喜欢研究 Python 同其他语言在设计和理论上的不同。因此在有些章节的最后，我会加上一点自己对 Python 以及其他语言的看法，我把这部分叫作“杂谈”。如果你对这些东西不感兴趣，跳过即可，因为这些并不是必读的。

Python术语表

我希望这本书不仅仅是关于 Python 的，也是关于 Python 的文化的。在过去 20 多年的交流中，Python 社区形成了它独有的行话和缩写。本书的最后有一部分叫"Python 术语表"，里面列出了在 Python 爱好者中具有特别意义的词句。

Python版本表

本书所有的代码都在 Python 3.4 里测试过，而且是应用最广的用 C 实现的 CPython 3.4。只有一个例外，在 13.4 节中的“Python 3.5 新引入的中缀运算符 `@`”附注栏里，我提到了新的 `@` 运算符，它只在 Python 3.5 里被支持。

凡是支持 Python 3.*x* 的解释器——包括 PyPy3 2.4.0——都可以运行书里的代码（PyPy3 2.4.0 其实已经支持 Python 3.2.5）。有一点需要注意的是，`yield from` 和 `asyncio` 只在 Python 3.3 或者更新的版本里才有。

几乎所有的代码稍做修改后都能在 Python 2.7 里运行，除了第 4 章中那些跟 Unicode 相关的例子，这是从 Python 3 出现以来就有的问题。

排版约定

本书使用了下列排版约定。

- 楷体

  表示新术语。

- 等宽字体（`constant width`）

  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 加粗等宽字体（**`constant width bold`**）

  表示应该由用户输入的命令或其他文本。

- 等宽斜体（*`Constant width italic`*）

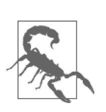
  表示应该由用户输入的值或根据上下文确定的值替换的文本。



该图标表示提示或建议。



该图标表示一般注记。



该图标表示警告或警示。

使用代码示例

书中的所有完整代码和大多数程序片段都可以从本书的 GitHub 代码库中获取（https://github.com/fluentpython/example-code）。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如：“*Fluent Python* by Luciano Ramalho (O'Reilly). Copyright 2015 Luciano Ramalho, 978-1-491-94600-8.”

Safari® Books Online



afari Books Onlinehttp://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那里找到本书的相关信息，包括勘误表、示例以及其他信息。本书的网站地址是：http://shop.oreilly.com/product/0636920032519.do

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

Josef Hartwig 设计的包豪斯国际象棋套装体现了最佳的设计理念：美观、简洁而清晰。有一位建筑师父亲，以及一位字体设计师弟弟，Guido van Rossum 设计出了一门经典的编程语言。我之所以热衷于教授 Python，也正是因为它的美观、简洁和清晰。

Alex Martelli 和 Anna Ravenscroft 是最先看到本书大纲的人，也是他们鼓励我把大纲交给 O'Reilly 出版社的。他们的书不但向我展示了地道的 Python 代码，还让我见识了什么才称得上是清晰、准确和有深度的技术写作。Alex 在 Stack Overflow 上的 5000 多个回答（http://stackoverflow.com/users/95810/alex-martelli）也体现了他对 Python 语言基础和正确用法的深入理解。

Martelli 和 Ravenscroft 同时也是本书的技术审稿人。除了他们之外，技术审稿人还有两位：Lennart Regebro 和 Leonardo Rochael。技术审稿团队里的每个人都至少有 15 年的 Python 经验，为许许多多具有广泛影响力的 Python 项目贡献过代码，并且跟社区里的其他开发者走得很近。审稿人一共提出了数百个修订、建议、问题和观点，为这本书做出了巨大贡献。另外，Victor Stinner 帮我审阅了第 18 章，他同时也是该章里提到的 `asyncio` 的维护者之一。在过去的几个月里能够跟他们合作，我感到很荣幸。

本书编辑 Meghan Blanchette 是一位出色的导师。她不但帮助我梳理整本书的结构、增强内容的连贯性，还为我指出哪里写得不够有趣，并且督促我及时交稿。Brian MacDonald 在 Meghan 休假的时候帮忙编辑了第三部分。跟他们以及 O'Reilly 的所有人打交道的过程都十分愉快。另外 Atlas 系统的开发和支持团队也很棒（Atlas 是 O'Reilly 的图书出版平台，我就是在这个平台上写作的）。

Mario Domenech Goulart 在看过本书第一次提前发行的版本后，提供了海量的详细建议。另外我还从 Dave Pawson、Elias Dorneles、Leonardo Alexandre Ferreira Leite、Bruce Eckel、J. S. Bueno、Rafael Gonçalves、Alex Chiaranda、Guto Maia、Lucas Vido 和 Lucas Brunialti 那里获得了宝贵的反馈。

几年来有很多人都在劝我写书，Rubens Prates、Aurelio Jargas、Rudá Moura 和 Rubens Altimari 这几位算是最有说服力的了。Mauricio Bussab 算得上带我入门的人，并且他让我有了第一次写书的尝试。Renzo Nuccitelli 毫不在乎这本书的写作可能会影响到我们合作的 python.pro.br 项目的进度，他从一开始就大力支持。

Python 巴西社区是一个集思广益、乐于分享且充满乐趣的地方。Python 巴西小组（https://groups.google.com/group/python-brasil）中有数千个人，每次的全国范围的会议都会把成百上千人聚集在一起。在我的 Python 爱好者成长之路上，对我影响最大的人有：Leonardo Rochael、Adriano Petrich、Daniel Vainsencher、Rodrigo RBP Pimentel、Bruno Gola、Leonardo Santagada、Jean Ferri、Rodrigo Senra、J. S. Bueno、David Kwast、Luiz Irber、 Osvaldo Santana、Fernando Masanori、Henrique Bastos、Gustavo Niemayer、Pedro Werneck、 Gustavo Barbieri、Lalo Martins、Danilo Bellini 和 Pedro Kroger。

Dorneles Tremea 是个非常棒的朋友（他很愿意花时间分享他的知识），他不但是很厉害的开发者，还是巴西 Python 协会中最鼓舞人心的领导人。可惜他过早离开了我们。

我的学生们同时也是我的老师，他们的问题、见解、反馈和那些富有创造性的回答教会了我很多。Érico Andrei 和 Simples Consultoria 让我头一次有机会集中精力做一名 Python 教师。

Martijn Faassen 是我的 Grok 导师，他同我分享了很多关于 Python 和尼安德特人的想法。Martijn 所做的事情，还有来自 Zope、Plone 和 Pyramid planets 的 Paul Everitt、Chris McDonough、Tres Seaver、Jim Fulton、Shane Hathaway、Lennart Regebro、Alan Runyan、Alexander Limi、Martijn Pieters 和 Godefroid Chapelle 等人所做的事情，在我事业发展的过程中起到了决定性的作用。多亏了 Zope 和第一波互联网浪潮，让我在 1998 年就开始从事 Python 相关的工作并以此为生。José Octavio Castro Neves 是我的搭档，我们在巴西开了第一家以 Python 业务为主的软件公司。

在更广阔的 Python 社区当中高手如云，我实在是没办法一一列出他们的名字。但是除了之前提到的之外，我还要感谢 Steve Holden、Raymond Hettinger、A.M. Kuchling、David Beazley、Fredrik Lundh、Doug Hellmann、Nick Coghlan、Mark Pilgrim、Martijn Pieters、Bruce Eckel、

Michele Simionato、Wesley Chun、Brandon Craig Rhodes、Philip Guo、Daniel Greenfeld、Audrey Roy 和 Brett Slatkin，感谢他们让我见识到更新更好的教授 Python 的方法。

我基本上是在家里的办公室和两个公共空间完成这本书的写作的。两个公共空间分别是 CoffeeLab 和 Garoa Hacker Clube。CoffeeLab（http://coffeelab.com.br）是位于巴西圣保罗 Vila Madalena 区的一个咖啡极客大本营。Garoa Hacker Clube（https://garoa.net.br）则是一个开放的黑客空间，任何人都可以在这里实验他们的新点子。

Garoa 社区还为我提供了灵感、基础设施和放松的环境，我想 Aleph 会喜欢这本书的。

我的母亲 Maria Lucia 和父亲 Jairo 一直都以各种方式支持我。我真希望我的父亲还健在并看到本书的出版，同时也为能与我的母亲分享这本书而感到开心。

在写这本书的 15 个月里，身为丈夫的我几乎一直在工作，我的妻子 Marta Mello 陪我一起熬过了这段日子。在这如同跑马拉松的写作过程中，她不但一直支持我，而且在我想要放弃的时候陪我一起渡过难关。

谢谢你们每一个人，谢谢你们做的每一件事。

电子书

扫描如下二维码，即可购买本书电子版。



第一部分　序幕

第 1 章　Python 数据模型

> Guido 对语言设计美学的深入理解让人震惊。我认识不少很不错的编程语言设计者，他们设计出来的东西确实很精彩，但是从来都不会有用户。Guido 知道如何在理论上做出一定妥协，设计出来的语言让使用者觉得如沐春风，这真是不可多得。[1]
>
> ——Jim Hugunin
> Jython 的作者，AspectJ 的作者之一，.NET DLR 架构师

[1]摘自“Story of Jython”（http://hugunin.net/story_of_jython.html），这是 *Jython Essentials*（Samuele Pedroni 和 Noel Rappin 著，O'Reilly 出版社，2002 年）一书的序。

Python 最好的品质之一是一致性。当你使用 Python 工作一会儿后，就会开始理解 Python 语言，并能正确猜测出对你来说全新的语言特征。

然而，如果你带着来自其他面向对象语言的经验进入 Python 的世界，会对 `len(colleciton)` 而不是 `collection.len()` 写法觉得不适。当你进一步理解这种不适感背后的原因之后，会发现这个原因，和它所代表的庞大的设计思想，是形成我们通常说的“Python 风格”（Pythonic）的关键。这种设计思想完全体现在 Python 的数据模型上，而数据模型所描述的 API，为使用最地道的语言特性来构建你自己的对象提供了工具。

数据模型其实是对 Python 框架的描述，它规范了这门语言自身构建模块的接口，这些模块包括但不限于序列、迭代器、函数、类和上下文管理器。

不管在哪种框架下写程序，都会花费大量时间去实现那些会被框架本身调用的方法， Python 也不例外。Python 解释器碰到特殊的句法时，会使用特殊方法去激活一些基本的对象操作，这些特殊方法的名字以两个下划线开头，以两个下划线结尾（例如 `__getitem__`）。比如 `obj[key]` 的背后就是 `__getitem__` 方法，为了能求得 `my_collection[key]` 的值，解释器实际上会调用 `my_collection.__getitem__(key)`。

这些特殊方法名能让你自己的对象实现和支持以下的语言构架，并与之

交互：

- 迭代
- 集合类
- 属性访问
- 运算符重载
- 函数和方法的调用
- 对象的创建和销毁
- 字符串表示形式和格式化
- 管理上下文（即 `with` 块）



magic 和 dunder

魔术方法（magic method）是特殊方法的昵称。有些 Python 开发者在提到 `__getitem__` 这个特殊方法的时候，会用诸如“下划线－下划线－ getitem”[2] 这种说法，但是显然这种说法会引起歧义，因为像 `__x` 这种命名在 Python 里还有其他含义，[3] 但是如果完整地说出“下划线－下划线－ getitem －下划线－下划线”，又会很麻烦。于是我跟着 Steve Holden，一位技术书作者和老师，学会了“双下－getitem”（dunder-getitem）这种说法。于是乎，特殊方法也叫双下方法（dunder method）。[4]

[2]即 under-under-getitme 的直译。——译者注

[3]注 3：详见 9.7 节。

[4]我是从 Steve Holden 那里第一次听说 dunder 这个说法的。根据维基百科的解释，Mark Johnson 和 Time Hochberg 是最早在书写中开始使用这个词的人（https://en.wikipedia.org/wiki/Reserved_word#Reserved_ranges）。那是 2002 年 9 月 26 日，他们两人在邮件列表里回复“__（双下划线）怎么念？”这个问题时提到了 dunder，最先回复的是 Johnson（https://mail.python.org/pipermail/python-list/2002-September/112991.html），11 分钟后

> Hochberg 也回复了（https://mail.python.org/pipermail/python-list/2002-September/114716.html）。

1.1　一摞Python风格的纸牌

接下来我会用一个非常简单的例子来展示如何实现 `__getitme__` 和 `__len__` 这两个特殊方法，通过这个例子我们也能见识到特殊方法的强大。

示例 1-1 里的代码建立了一个纸牌类。

示例 **1-1**　一摞有序的纸牌

```
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                                        for rank in self.ranks]

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, position):
        return self._cards[position]
```

首先，我们用 `collections.namedtuple` 构建了一个简单的类来表示一张纸牌。自 Python 2.6 开始，`namedtuple` 就加入到 Python 里，用以构建只有少数属性但是没有方法的对象，比如数据库条目。如下面这个控制台会话所示，利用 `namedtuple`，我们可以很轻松地得到一个纸牌对象：

```
>>> beer_card = Card('7', 'diamonds')
>>> beer_card
Card(rank='7', suit='diamonds')
```

当然，我们这个例子主要还是关注 FrenchDeck 这个类，它既短小又精悍。首先，它跟任何标准 Python 集合类型一样，可以用 len() 函数来查看一叠牌有多少张：

```
>>> deck = FrenchDeck()
>>> len(deck)
52
```

从一叠牌中抽取特定的一张纸牌，比如说第一张或最后一张，是很容易的：deck[0] 或 deck[-1]。这都是由 __getitem__ 方法提供的：

```
>>> deck[0]
Card(rank='2', suit='spades')
>>> deck[-1]
Card(rank='A', suit='hearts')
```

我们需要单独写一个方法用来随机抽取一张纸牌吗？没必要，Python 已经内置了从一个序列中随机选出一个元素的函数 random.choice，我们直接把它用在这一摞纸牌实例上就好：

```
>>> from random import choice
>>> choice(deck)
Card(rank='3', suit='hearts')
>>> choice(deck)
Card(rank='K', suit='spades')
>>> choice(deck)
Card(rank='2', suit='clubs')
```

现在已经可以体会到通过实现特殊方法来利用 Python 数据模型的两个好处。

- 作为你的类的用户，他们不必去记住标准操作的各式名称（“怎么得到元素的总数？是 .size() 还是 .length() 还是别的什么？”）。

- 可以更加方便地利用 Python 的标准库，比如 random.choice 函数，从而不用重新发明轮子。

而且好戏还在后面。

因为 `__getitem__` 方法把 `[]` 操作交给了 `self._cards` 列表，所以我们的 `deck` 类自动支持切片（slicing）操作。下面列出了查看一摞牌最上面 3 张和只看牌面是 A 的牌的操作。其中第二种操作的具体方法是，先抽出索引是 12 的那张牌，然后每隔 13 张牌拿 1 张：

```
>>> deck[:3]
[Card(rank='2', suit='spades'), Card(rank='3', suit='spades'),
Card(rank='4', suit='spades')]
>>> deck[12::13]
[Card(rank='A', suit='spades'), Card(rank='A', suit='diamonds'),
Card(rank='A', suit='clubs'), Card(rank='A', suit='hearts')]
```

另外，仅仅实现了 `__getitem__` 方法，这一摞牌就变成可迭代的了：

```
>>> for card in deck: # doctest: +ELLIPSIS
...   print(card)
Card(rank='2', suit='spades')
Card(rank='3', suit='spades')
Card(rank='4', suit='spades')
...
```

反向迭代也没关系：

```
>>> for card in reversed(deck): # doctest: +ELLIPSIS
...   print(card)
Card(rank='A', suit='hearts')
Card(rank='K', suit='hearts')
Card(rank='Q', suit='hearts')
...
```



doctest 中的省略

为了尽可能保证书中的 Python 控制台会话内容的正确性，这些内容都是直接从 doctest 里摘录的。在测试中，如果可能的输出过长的话，那么过长的内容就会被如上面例子的最后一行的省略号（`...`）所替代。此时就需要 `#doctest: +ELLIPSIS` 这个指令来保证 doctest 能够通过。要是你自己照着书中例子在控制台中敲代码，可以略过这一指令。

迭代通常是隐式的，譬如说一个集合类型没有实现 `__contains__` 方法，那么 `in` 运算符就会按顺序做一次迭代搜索。于是，`in` 运算符可以用在我们的 `FrenchDeck` 类上，因为它是可迭代的：

```
>>> Card('Q', 'hearts') in deck
True
>>> Card('7', 'beasts') in deck
False
```

那么排序呢？我们按照常规，用点数来判定扑克牌的大小，2 最小、A 最大；同时还要加上对花色的判定，黑桃最大、红桃次之、方块再次、梅花最小。下面就是按照这个规则来给扑克牌排序的函数，梅花 2 的大小是 `0`，黑桃 A 是 `51`：

```
suit_values = dict(spades=3, hearts=2, diamonds=1, clubs=0)
def spades_high(card):
    rank_value = FrenchDeck.ranks.index(card.rank)
    return rank_value * len(suit_values) + suit_values[card.suit]
```

有了 `spades_high` 函数，就能对这摞牌进行升序排序了：

```
>>> for card in sorted(deck, key=spades_high): # doctest: +ELLIPSIS
...      print(card)
Card(rank='2', suit='clubs')
Card(rank='2', suit='diamonds')
Card(rank='2', suit='hearts')
... (46 cards ommitted)
Card(rank='A', suit='diamonds')
Card(rank='A', suit='hearts')
Card(rank='A', suit='spades')
```

虽然 `FrenchDeck` 隐式地继承了 `object` 类，[5] 但功能却不是继承而来的。我们通过数据模型和一些合成来实现这些功能。通过实现 `__len__` 和 `__getitem__` 这两个特殊方法，`FrenchDeck` 就跟一个 Python 自有的序列数据类型一样，可以体现出 Python 的核心语言特性（例如迭代和切片）。同时这个类还可以用于标准库中诸如 `random.choice`、`reversed` 和 `sorted` 这些函数。另外，对合成的运用使得 `__len__` 和 `__getitem__` 的具体实现可以代理给 `self._cards` 这个 Python 列表（即 `list` 对象）。

[5]在 Python 2 中，对 `object` 的继承需要显式地写为 `FrenchDeck(object)`；而在 Python 3 中，这个继承关系是默认的。



如何洗牌

按照目前的设计，`FrenchDeck` 是不能洗牌的，因为这摞牌是不可变的（immutable）：卡牌和它们的位置都是固定的，除非我们破坏这个类的封装性，直接对 `_cards` 进行操作。第 11 章会讲到，其实只需要一行代码来实现 `__setitem__` 方法，洗牌功能就不是问题了。

1.2 如何使用特殊方法

首先明确一点，特殊方法的存在是为了被 Python 解释器调用的，你自己并不需要调用它们。也就是说没有 `my_object.__len__()` 这种写法，而应该使用 `len(my_object)`。在执行 `len(my_object)` 的时候，如果 `my_object` 是一个自定义类的对象，那么 Python 会自己去调用其中由你实现的 `__len__` 方法。

然而如果是 Python 内置的类型，比如列表（`list`）、字符串（`str`）、字节序列（`bytearray`）等，那么 CPython 会抄个近路，`__len__` 实际上会直接返回 `PyVarObject` 里的 `ob_size` 属性。`PyVarObject` 是表示内存中长度可变的内置对象的 C 语言结构体。直接读取这个值比调用一个方法要快很多。

很多时候，特殊方法的调用是隐式的，比如 `for i in x:` 这个语句，背后其实用的是 `iter(x)`，而这个函数的背后则是 `x.__iter__()` 方法。当然前提是这个方法在 `x` 中被实现了。

通常你的代码无需直接使用特殊方法。除非有大量的元编程存在，直接调用特殊方法的频率应该远远低于你去实现它们的次数。唯一的例外可能是 `__init__` 方法，你的代码里可能经常会用到它，目的是在你自己的子类的 `__init__` 方法中调用超类的构造器。

通过内置的函数（例如 `len`、`iter`、`str`，等等）来使用特殊方法是最好的选择。这些内置函数不仅会调用特殊方法，通常还提供额外的好处，而且对于内置的类来说，它们的速度更快。14.12 节中有详细的例子。

不要自己想当然地随意添加特殊方法，比如 `__foo__` 之类的，因为虽然现在这个名字没有被 Python 内部使用，以后就不一定了。

1.2.1 模拟数值类型

利用特殊方法，可以让自定义对象通过加号“+”（或是别的运算符）进行运算。第 13 章对此有详细的介绍，现在只是借用这个例子来展示特

殊方法的使用。

我们来实现一个二维向量（vector）类，这里的向量就是欧几里得几何中常用的概念，常在数学和物理中使用的那个（见图 1-1）。

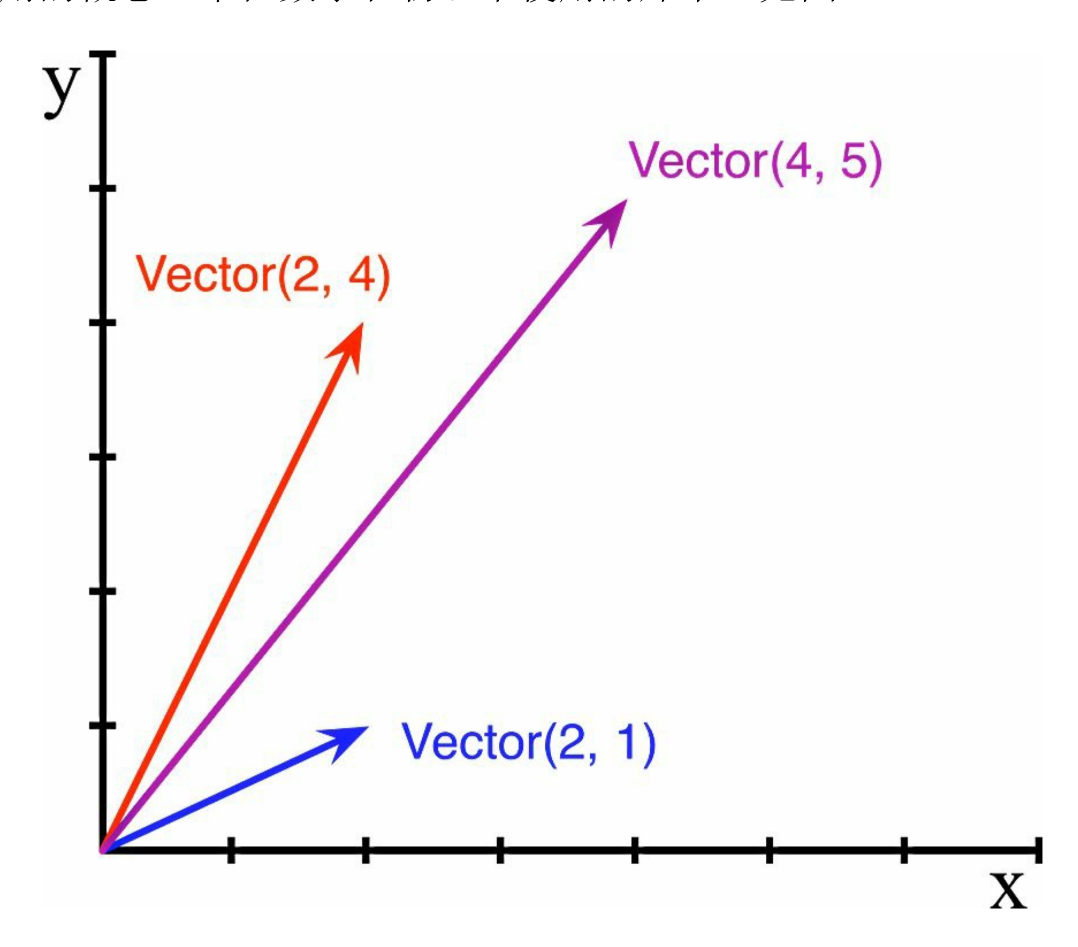


图 **1-1**：一个二维向量加法的例子，**`Vector(2,4) + Vextor(2,1) = Vector(4,5)`**

Python 内置的 `complex` 类可以用来表示二维向量，但我们这个自定义的类可以扩展到 *n* 维向量，详见第 14 章。

为了给这个类设计 API，我们先写个模拟的控制台会话来做 doctest。下面这一段代码就是图 1-1 所示的向量加法：

```
>>> v1 = Vector(2, 4)
>>> v2 = Vector(2, 1)
>>> v1 + v2
Vector(4, 5)
```

注意其中的 + 运算符所得到的结果也是一个向量，而且结果能被控制台友好地打印出来。

abs 是一个内置函数，如果输入是整数或者浮点数，它返回的是输入值的绝对值；如果输入是复数（complex number），那么返回这个复数的模。为了保持一致性，我们的 API 在碰到 abs 函数的时候，也应该返回该向量的模：

```
>>> v = Vector(3, 4)
>>> abs(v)
5.0
```

我们还可以利用 * 运算符来实现向量的标量乘法（即向量与数的乘法，得到的结果向量的方向与原向量一致[6]，模变大）：

> [6]如果向量与负数相乘，得到的结果向量的方向与原向量相反。——编者注

```
>>> v * 3
Vector(9, 12)
>>> abs(v * 3)
15.0
```

示例 1-2 包含了一个 Vector 类的实现，上面提到的操作在代码里是用这些特殊方法实现的：__repr__、__abs__、__add__ 和 __mul__。

示例 **1-2**　一个简单的二维向量类

```
from math import hypot

class Vector:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __repr__(self):
        return 'Vector(%r, %r)' % (self.x, self.y)

    def __abs__(self):
        return hypot(self.x, self.y)

```

```
    def __bool__(self):
        return bool(abs(self))

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)
```

虽然代码里有 6 个特殊方法，但这些方法（除了 `__init__`）并不会在这个类自身的代码中使用。即便其他程序要使用这个类的这些方法，也不会直接调用它们，就像我们在上面的控制台对话中看到的。上文也提到过，一般只有 Python 的解释器会频繁地直接调用这些方法。接下来看看每个特殊方法的实现。

1.2.2 字符串表示形式

Python 有一个内置的函数叫 `repr`，它能把一个对象用字符串的形式表达出来以便辨认，这就是“字符串表示形式”。`repr` 就是通过 `__repr__` 这个特殊方法来得到一个对象的字符串表示形式的。如果没有实现 `__repr__`，当我们在控制台里打印一个向量的实例时，得到的字符串可能会是 `<Vector object at 0x10e100070>`。

交互式控制台和调试程序（debugger）用 `repr` 函数来获取字符串表示形式；在老的使用 `%` 符号的字符串格式中，这个函数返回的结果用来代替 `%r` 所代表的对象；同样，`str.format` 函数所用到的新式字符串格式化语法（https://docs.python.org/2/library/string.html#format-string-syntax）也是利用了 `repr`，才把 `!r` 字段变成字符串。

> `%` 和 `str.format` 这两种格式化字符串的手段在本书中都会使用。其实整个 Python 社区都在同时使用这两种方法。个人来讲，我越来越喜欢 `str.format` 了，但是 Python 程序员更喜欢简单的 `%`。因此，这两种形式并存的情况还会持续下去。

在 `__repr__` 的实现中，我们用到了 `%r` 来获取对象各个属性的标准字

符串表示形式——这是个好习惯，它暗示了一个关键：Vector(1, 2) 和 Vector('1', '2') 是不一样的，后者在我们的定义中会报错，因为向量对象的构造函数只接受数值，不接受字符串 [7]。

> [7]实际上，Vector 的构造函数接受字符串。而且，对于使用字符串构造的 Vector，这 6 个特殊方法中，只有 __abs__ 和 __bool__ 会报错。此外，1.2.4 节定义的 __bool__ 不会报错。——编者注

__repr__ 所返回的字符串应该准确、无歧义，并且尽可能表达出如何用代码创建出这个被打印的对象。因此这里使用了类似调用对象构造器的表达形式（比如 Vector(3, 4) 就是个例子）。

__repr__ 和 __str__ 的区别在于，后者是在 str() 函数被使用，或是在用 print 函数打印一个对象的时候才被调用的，并且它返回的字符串对终端用户更友好。

如果你只想实现这两个特殊方法中的一个，__repr__ 是更好的选择，因为如果一个对象没有 __str__ 函数，而 Python 又需要调用它的时候，解释器会用 __repr__ 作为替代。

"Difference between __str__ and __repr__ in Python"（http://stackoverflow.com/questions/1436703/difference-between-str-and-repr-in-python）是 Stack Overflow 上的一个问题，Python 程序员 Alex Martelli 和 Martijn Pieters 的回答很精彩。

1.2.3　算术运算符

通过 __add__ 和 __mul__，示例 1-2 为向量类带来了 + 和 * 这两个算术运算符。值得注意的是，这两个方法的返回值都是新创建的向量对象，被操作的两个向量（self 或 other）还是原封不动，代码里只是读取了它们的值而已。中缀运算符的基本原则就是不改变操作对象，而是产出一个新的值。第 13 章会谈到更多这方面的问题。

示例 1-2 只实现了数字做乘数、向量做被乘数的运算，乘法的交换律则被忽略了。在第 13 章里，我们将利用 __rmul__ 解决

这个问题。

1.2.4　自定义的布尔值

尽管 Python 里有 bool 类型，但实际上任何对象都可以用于需要布尔值的上下文中（比如 if 或 while 语句，或者 and、or 和 not 运算符）。为了判定一个值 x 为真还是为假，Python 会调用 bool(x)，这个函数只能返回 True 或者 False。

默认情况下，我们自己定义的类的实例总被认为是真的，除非这个类对 __bool__ 或者 __len__ 函数有自己的实现。bool(x) 的背后是调用 x.__bool__() 的结果；如果不存在 __bool__ 方法，那么 bool(x) 会尝试调用 x.__len__()。若返回 0，则 bool 会返回 False；否则返回 True。

我们对 __bool__ 的实现很简单，如果一个向量的模是 0，那么就返回 False，其他情况则返回 True。因为 __bool__ 函数的返回类型应该是布尔型，所以我们通过 bool(abs(self)) 把模值变成了布尔值。

在 Python 标准库的文档中，有一节叫作“Built-in Types”（https://docs.python.org/3/library/stdtypes.html#truth），其中规定了真值检验的标准。通过实现 __bool__，你定义的对象就可以与这个标准保持一致。



如果想让 Vector.__bool__ 更高效，可以采用这种实现：

```
def __bool__(self):
    return bool(self.x or self.y)
```

它不那么易读，却能省掉从 abs 到 __abs__ 到平方再到平方根这些中间步骤。通过 bool 把返回类型显式转换为布尔值是为了符合 __bool__ 对返回值的规定，因为 or 运算符可能会返回 x 或者 y 本身的值：若 x 的值等价于真，则 or 返回 x 的值；否则返回 y 的

值。

1.3 特殊方法一览

Python 语言参考手册中的“Data Model”（https://docs.python.org/3/reference/datamodel.html）一章列出了 83 个特殊方法的名字，其中 47 个用于实现算术运算、位运算和比较操作。

表 1-1 和表 1-2 列出了这些方法的概况。



这些表并没有完全按照官方文档分组。

表**1-1**：跟运算符无关的特殊方法

类别	方法名
字符串 / 字节序列表示形式	`__repr__`、`__str__`、`__format__`、`__bytes__`
数值转换	`__abs__`、`__bool__`、`__complex__`、`__int__`、`__float__`、`__hash__`、`__index__`
集合模拟	`__len__`、`__getitem__`、`__setitem__`、`__delitem__`、`__contains__`
迭代枚举	`__iter__`、`__reversed__`、`__next__`
可调用模拟	`__call__`

上下文管理	`__enter__`、`__exit__`
实例创建和销毁	`__new__`、`__init__`、`__del__`
属性管理	`__getattr__`、`__getattribute__`、`__setattr__`、`__delattr__`、`__dir__`
属性描述符	`__get__`、`__set__`、`__delete__`
跟类相关的服务	`__prepare__`、`__instancecheck__`、`__subclasscheck__`

表**1-2**：跟运算符相关的特殊方法

类别	方法名和对应的运算符
一元运算符	`__neg__ -`、`__pos__ +`、`__abs__ abs()`
众多比	

较运算符	__lt__ <、__le__ <=、__eq__ ==、__ne__ !=、__gt__ >、__ge__ >=
算术运算符	__add__ +、__sub__ -、__mul__ *、__truediv__ /、__floordiv__ //、__mod__ %、__divmod divmod()、__pow__ ** 或pow()、__round__ round()
反向算术运算符	__radd__、__rsub__、__rmul__、__rtruediv__、__rfloordiv__、__rmod__、__rdivmod__、__
增量赋值算术运算符	__iadd__、__isub__、__imul__、__itruediv__、__ifloordiv__、__imod__、__ipow__
位运算符	__invert__ ~、__lshift__ <<、__rshift__ >>、__and__ &、__or__ |、__xor__ ^
反向位运算符	__rlshift__、__rrshift__、__rand__、__rxor__、__ror__

增量赋值位运算符	`__ilshift__`、`__irshift__`、`__iand__`、`__ixor__`、`__ior__`

当交换两个操作数的位置时，就会调用反向运算符（`b * a` 而不是 `a * b`）。增量赋值运算符则是一种把中缀运算符变成赋值运算的捷径（`a = a * b` 就变成了 `a *= b`）。第 13 章会对这两者作出详细解释。

1.4　为什么len不是普通方法

我在 2013 年问核心开发者 Raymond Hettinger 这个问题时，他用"Python 之禅"（https://www.python.org/doc/humor/#the-zen-of-python）里的原话回答了我："实用胜于纯粹。"在 1.2 节里我提到过，如果 x 是一个内置类型的实例，那么 len(x) 的速度会非常快。背后的原因是 CPython 会直接从一个 C 结构体里读取对象的长度，完全不会调用任何方法。获取一个集合中元素的数量是一个很常见的操作，在 str、list、memoryview 等类型上，这个操作必须高效。

换句话说，len 之所以不是一个普通方法，是为了让 Python 自带的数据结构可以走后门，abs 也是同理。但是多亏了它是特殊方法，我们也可以把 len 用于自定义数据类型。这种处理方式在保持内置类型的效率和保证语言的一致性之间找到了一个平衡点，也印证了"Python 之禅"中的另外一句话："不能让特例特殊到开始破坏既定规则。"

> 如果把 abs 和 len 都看作一元运算符的话，你也许更能接受它们——虽然看起来像面向对象语言中的函数，但实际上又不是函数。有一门叫作 ABC 的语言是 Python 的直系祖先，它内置了一个 # 运算符，当你写出 #s 的时候，它的作用跟 len 一样。如果写成 x#s 这样的中缀运算符的话，那么它的作用是计算 s 中 x 出现的次数。在 Python 里对应的写法是 s.count(x)。注意这里的 s 是一个序列类型。

1.5 本章小结

通过实现特殊方法，自定义数据类型可以表现得跟内置类型一样，从而让我们写出更具表达力的代码——或者说，更具 Python 风格的代码。

Python 对象的一个基本要求就是它得有合理的字符串表示形式，我们可以通过 `__repr__` 和 `__str__` 来满足这个要求。前者方便我们调试和记录日志，后者则是给终端用户看的。这就是数据模型中存在特殊方法 `__repr__` 和 `__str__` 的原因。

对序列数据类型的模拟是特殊方法用得最多的地方，这一点在 `FrenchDeck` 类的示例中有所展现。在第 2 章中，我们会着重介绍序列数据类型，然后在第 10 章中，我们会把 `Vector` 类扩展成一个多维的数据类型，通过这个练习你将有机会实现自定义的序列。

Python 通过运算符重载这一模式提供了丰富的数值类型，除了内置的那些之外，还有 `decimal.Decimal` 和 `fractions.Fraction`。这些数据类型都支持中缀算术运算符。在第 13 章中，我们还会通过对 `Vector` 类的扩展来学习如何实现这些运算符，当然还会提到如何让运算符满足交换律和增强赋值。

Python 数据模型的特殊方法还有很多，本书会涵盖其中的绝大部分，探讨如何使用和实现它们。

1.6　延伸阅读

对本章内容和本书主题来说，Python 语言参考手册里的“Data Model”一章（https://docs.python.org/3/reference/datamodel.html）是最符合规范的知识来源。

Alex Martelli 的《Python 技术手册（第 2 版）》对数据模型的讲解很精彩。我写这本书的时候，《Python 技术手册》的最新版本是 2006 年出版的，书里用的还是 Python 2.5，但是 Python 关于数据模型的概念并没有太大的变化，而书中 Martelli 对属性访问机制的描述，应该是除了 CPython 中的 C 源码之外在这方面最权威的解释。Martelli 还是 Stack Overflow 上的高产贡献者，在他名下差不多有 5000 条答案，你也可以去他的 Stack Overflow 主页（http://stackoverflow.com/users/95810/alex-martelli）上看看。

David Beazley 著有两本基于 Python 3 的书，其中对数据模型进行了详尽的介绍。一本是《Python 参考手册（第 4 版）》[8]，另一本是与 Brian K. Jones 合著的《Python Cookbook（第 3 版）中文版》。

[8]该书已由人民邮电出版社出版，书号：978-7-115-24259-4。——编者注

由 Gregor Kiczales、Jim des Rivieres 和 Daniel G. Bobrow 合著的 *The Art of the Metaobject Protocol*（又称 *AMOP*，MIT 出版社，1991 年）一书解释了元对象协议（metaobject protocol，MOP）的概念，而 Python 数据模型便是对这一概念的一种阐释。

杂谈

数据模型还是对象模型

Python 文档里总是用“Python 数据模型”这种说法，而大多数作者提到这个概念的时候会说“Python 对象模型”。Alex Martelli 的《Python 技术手册（第 2 版）》和 David Beazley 的《Python 参考手册（第 4 版）》是这个领域中最好的两本书，但是他们也总说“Python 对象模型”。维基百科中对象模型的第一个定义（http://en.wikipedia.org/wiki/Object_model）是：计算机编程语言中

对象的属性。这正好是“Python 数据模型”所要描述的概念。我在本书中一直都会用“数据模型”这个词，首先是因为在 Python 文档里对这个词有偏爱，另外一个原因是 Python 语言参考手册中与这里讨论的内容最相关的一章的标题就是“数据模型”（https://docs.python.org/3/reference/datamodel.html）。

魔术方法

在 Ruby 中也有类似“特殊方法”的概念，但是 Ruby 社区称之为“魔术方法”，而实际上 Python 社区里也有不少人用的是后者。而我恰恰认为“特殊方法”是“魔术方法”的对立面。Python 和 Ruby 都利用了这个概念来提供丰富的元对象协议，这不是魔术，而是让语言的用户和核心开发者拥有并使用同样的工具。

考虑一下 JavaScript，情况就正好反过来了。JavaScript 中的对象有不透明的魔术般的特性，而你无法在自定义的对象中模拟这些行为。比如在 JavaScript 1.8.5 中，用户的自定义对象不能有只读属性，然而不少 JavaScript 的内置对象却可以有。因此在 JavaScript 中，只读属性是“魔术”般的存在，对于普通的 JavaScript 用户而言，它就像超能力一样。2009 年推出的 ECMAScript 5.1 才让用户可以定义只读属性。JavaScript 中跟元对象协议有关的部分一直在进化，但由于历史原因，这方面它还是赶不上 Python 和 Ruby。

元对象

The Art of the Metaobject Protocal（*AMOP*）是我最喜欢的计算机图书的标题。客观来说，元对象协议这个词对我们学习Python 数据模型是有帮助的。元对象所指的是那些对建构语言本身来讲很重要的对象，以此为前提，协议也可以看作接口。也就是说，元对象协议是对象模型的同义词，它们的意思都是构建核心语言的 API。

一套丰富的元对象协议能让我们对语言进行扩展，让它支持新的编程范式。*AMOP* 的第一作者 Gregor Kiczales 后来成为面向方面编程的先驱，他写出了一个 Java 扩展叫 AspectJ，用来实现他对面向方面编程的理念。其实在 Python 这样的动态语言里，更容易实现面向方面编程。现在已经有几个 Python 框架在做这件事情了，其中最重要的是 `zope.interface`（http://docs.zope.org/zope.interface/）。第

11 章的延伸阅读里会谈到它。

第二部分　数据结构

第 2 章　序列构成的数组

> 你可能注意到了，之前提到的几个操作可以无差别地应用于文本、列表和表格上。我们把文本、列表和表格叫作数据火车……`FOR` 命令通常能作用于数据火车上。[1]
>
> ——Geurts、Meertens 和 Pemberton
> *ABC Programmer's Handbook*

[1]Leo Geurts, Lambert Meertens, and Steven Pemberton, *ABC Programmer's Handbook*, p. 8.

在创造 Python 以前，Guido 曾为 ABC 语言贡献过代码。ABC 语言是一个致力于为初学者设计编程环境的长达 10 年的研究项目，其中很多点子在现在看来都很有 Python 风格：序列的泛型操作、内置的元组和映射类型、用缩进来架构的源码、无需变量声明的强类型，等等。Python 对开发者如此友好，根源就在这里。

Python 也从 ABC 那里继承了用统一的风格去处理序列数据这一特点。不管是哪种数据结构，字符串、列表、字节序列、数组、XML 元素，抑或是数据库查询结果，它们都共用一套丰富的操作：迭代、切片、排序，还有拼接。

深入理解 Python 中的不同序列类型，不但能让我们避免重新发明轮子，它们的 API 还能帮助我们把自己定义的 API 设计得跟原生的序列一样，或者是跟未来可能出现的序列类型保持兼容。

本章讨论的内容几乎可以应用到所有的序列类型上，从我们熟悉的 `list`，到 Python 3 中特有的 `str` 和 `bytes`。我还会特别提到跟列表、元组、数组以及队列有关的话题。但是 Unicode 字符串和字节序列这方面的内容被放在了第 4 章。另外这里讨论的数据结构都是 Python 中现成可用的，如果你想知道怎样创建自己的序列类型，那得等到第 10 章。

2.1 内置序列类型概览

Python 标准库用 C 实现了丰富的序列类型，列举如下。

容器序列

　　`list`、`tuple` 和 `collections.deque` 这些序列能存放不同类型的数据。

扁平序列

　　`str`、`bytes`、`bytearray`、`memoryview` 和 `array.array`，这类序列只能容纳一种类型。

容器序列存放的是它们所包含的任意类型的对象的引用，而扁平序列里存放的是值而不是引用。换句话说，扁平序列其实是一段连续的内存空间。由此可见扁平序列其实更加紧凑，但是它里面只能存放诸如字符、字节和数值这种基础类型。

序列类型还能按照能否被修改来分类。

可变序列

　　`list`、`bytearray`、`array.array`、`collections.deque` 和 `memoryview`。

不可变序列

　　`tuple`、`str` 和 `bytes`。

图 2-1 显示了可变序列（`MutableSequence`）和不可变序列（`Sequence`）的差异，同时也能看出前者从后者那里继承了一些方法。虽然内置的序列类型并不是直接从 `Sequence` 和 `MutableSequence` 这两个抽象基类（Abstract Base Class，ABC）继承而来的，但是了解这些基类可以帮助我们总结出那些完整的序列类型包含了哪些功能。

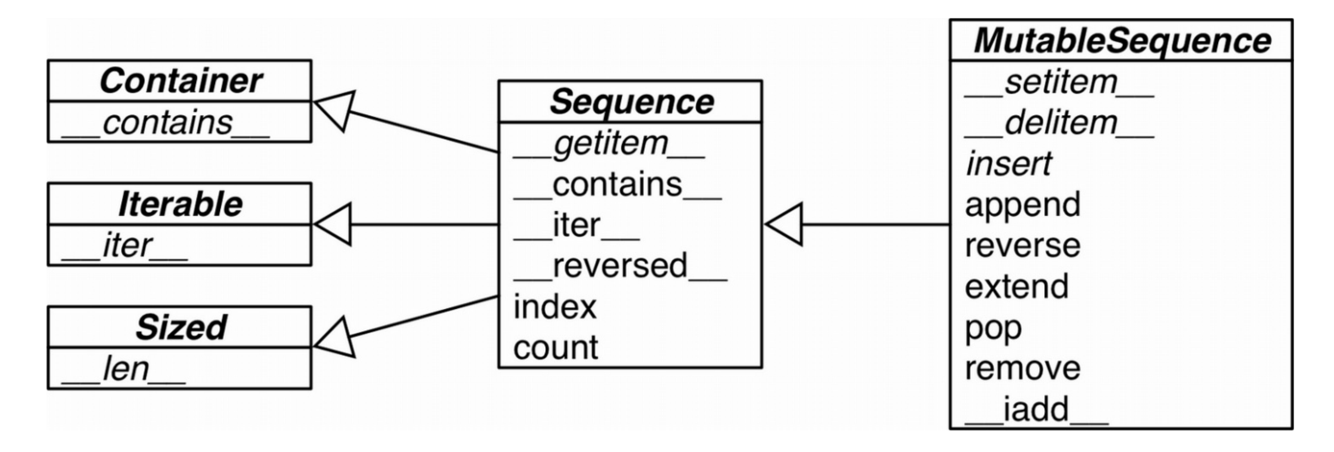


图 **2-1**：这个 **UML** 类图列举了 **`collections.abc`** 中的几个类（超类在左边，箭头从子类指向超类，斜体名称代表抽象类和抽象方法）

通过记住这些类的共有特性，把可变与不可变序列或是容器与扁平序列的概念融会贯通，在探索并学习新的序列类型时，你会更加得心应手。

最重要也最基础的序列类型应该就是列表（`list`）了。`list` 是一个可变序列，并且能同时存放不同类型的元素。作为这本书的读者，我想你应该对它很了解了，因此让我们直接开始讨论列表推导（list comprehension）吧。列表推导是一种构建列表的方法，它异常强大，然而由于相关的句法比较晦涩，人们往往不愿意去用它。掌握列表推导还可以为我们打开生成器表达式（generator expression）的大门，后者具有生成各种类型的元素并用它们来填充序列的功能。下一节就来看看这两个概念。

2.2　列表推导和生成器表达式

列表推导是构建列表（`list`）的快捷方式，而生成器表达式则可以用来创建其他任何类型的序列。如果你的代码里并不经常使用它们，那么很可能你错过了许多写出可读性更好且更高效的代码的机会。

如果你对我说的"更具可读性"持怀疑态度的话，别急着下结论，我马上就能说服你。

> 很多 Python 程序员都把列表推导（list comprehension）简称为 listcomps，生成式表达器（generator expression）则称为 genexps。我有时也会这么用。

2.2.1　列表推导和可读性

先来个小测试，你觉得示例 2-1 和示例 2-2 中的代码，哪个更容易读懂？

示例 **2-1**　把一个字符串变成 Unicode 码位的列表

```
>>> symbols = '$¢£¥€¤'
>>> codes = []
>>> for symbol in symbols:
...     codes.append(ord(symbol))
...
>>> codes
[36, 162, 163, 165, 8364, 164]
```

示例 **2-2**　把字符串变成 Unicode 码位的另外一种写法

```
>>> symbols = '$¢£¥€¤'
>>> codes = [ord(symbol) for symbol in symbols]
>>> codes
[36, 162, 163, 165, 8364, 164]
```

虽说任何学过一点 Python 的人应该都能看懂示例 2-1，但是我觉得如果

学会了列表推导的话，示例 2-2 读起来更方便，因为这段代码的功能从字面上就能轻松地看出来。

`for` 循环可以胜任很多任务：遍历一个序列以求得总数或挑出某个特定的元素、用来计算总和或是平均数，还有其他任何你想做的事情。在示例 2-1 的代码里，它被用来新建一个列表。

另一方面，列表推导也可能被滥用。以前看到过有的 Python 代码用列表推导来重复获取一个函数的副作用。通常的原则是，只用列表推导来创建新的列表，并且尽量保持简短。如果列表推导的代码超过了两行，你可能就要考虑是不是得用 `for` 循环重写了。就跟写文章一样，并没有什么硬性的规则，这个度得你自己把握。



句法提示

Python 会忽略代码里 `[]`、`{}` 和 `()` 中的换行，因此如果你的代码里有多行的列表、列表推导、生成器表达式、字典这一类的，可以省略不太好看的续行符 `\`。

列表推导不会再有变量泄漏的问题

Python 2.*x* 中，在列表推导中 `for` 关键词之后的赋值操作可能会影响列表推导上下文中的同名变量。像下面这个 Python 2.7 控制台对话：

```
Python 2.7.6 (default, Mar 22 2014, 22:59:38)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 'my precious'
>>> dummy = [x for x in 'ABC']
>>> x
'C'
```

如你所见，`x` 原本的值被取代了，但是这种情况在 Python 3 中是不会出现的。

列表推导、生成器表达式，以及同它们很相似的集合（set）推导和字典（dict）推导，在 Python 3 中都有了自己的局部作用域，就像函数似的。表达式内部的变量和赋值只在局部起作用，表达式的上下文里的同名变量还可以被正常引用，局部变量并不会影响到它们。

这是Python 3 代码：

```
>>> x = 'ABC'
>>> dummy = [ord(x) for x in x]
>>> x ❶
'ABC'
>>> dummy ❷
[65, 66, 67]
>>>
```

❶ x 的值被保留了。

❷ 列表推导也创建了正确的列表。

列表推导可以帮助我们把一个序列或是其他可迭代类型中的元素过滤或是加工，然后再新建一个列表。Python 内置的 filter 和 map 函数组合起来也能达到这一效果，但是可读性上打了不小的折扣。

2.2.2　列表推导同**filter**和**map**的比较

filter 和 map 合起来能做的事情，列表推导也可以做，而且还不需要借助难以理解和阅读的 lambda 表达式。详见示例 2-3。

示例 **2-3**　用列表推导和 map/filter 组合来创建同样的表单

```
>>> symbols = '$¢£¥€¤'
>>> beyond_ascii = [ord(s) for s in symbols if ord(s) > 127]
>>> beyond_ascii
[162, 163, 165, 8364, 164]
>>> beyond_ascii = list(filter(lambda c: c > 127, map(ord, symbols)))
>>> beyond_ascii
[162, 163, 165, 8364, 164]
```

我原以为 `map/filter` 组合起来用要比列表推导快一些，Alex Martelli 却说不一定——至少在上面这个例子中不一定。在本书的代码仓库（https://github.com/fluentpython/example-code）中有名为 02-array-seq/listcomp_speed.py（https://github.com/fluentpython/example-code/blob/master/02-array-seq/listcomp_speed.py）的脚本，代码中有这两个方法的效率的比较。

第 5 章会更详细地讨论 `map` 和 `filter`。下面就来看看如何用列表推导来计算笛卡儿积：两个或以上的列表中的元素对构成元组，这些元组构成的列表就是笛卡儿积。

2.2.3　笛卡儿积

如前所述，用列表推导可以生成两个或以上的可迭代类型的笛卡儿积。笛卡儿积是一个列表，列表里的元素是由输入的可迭代类型的元素对构成的元组，因此笛卡儿积列表的长度等于输入变量的长度的乘积，如图 2-2 所示。

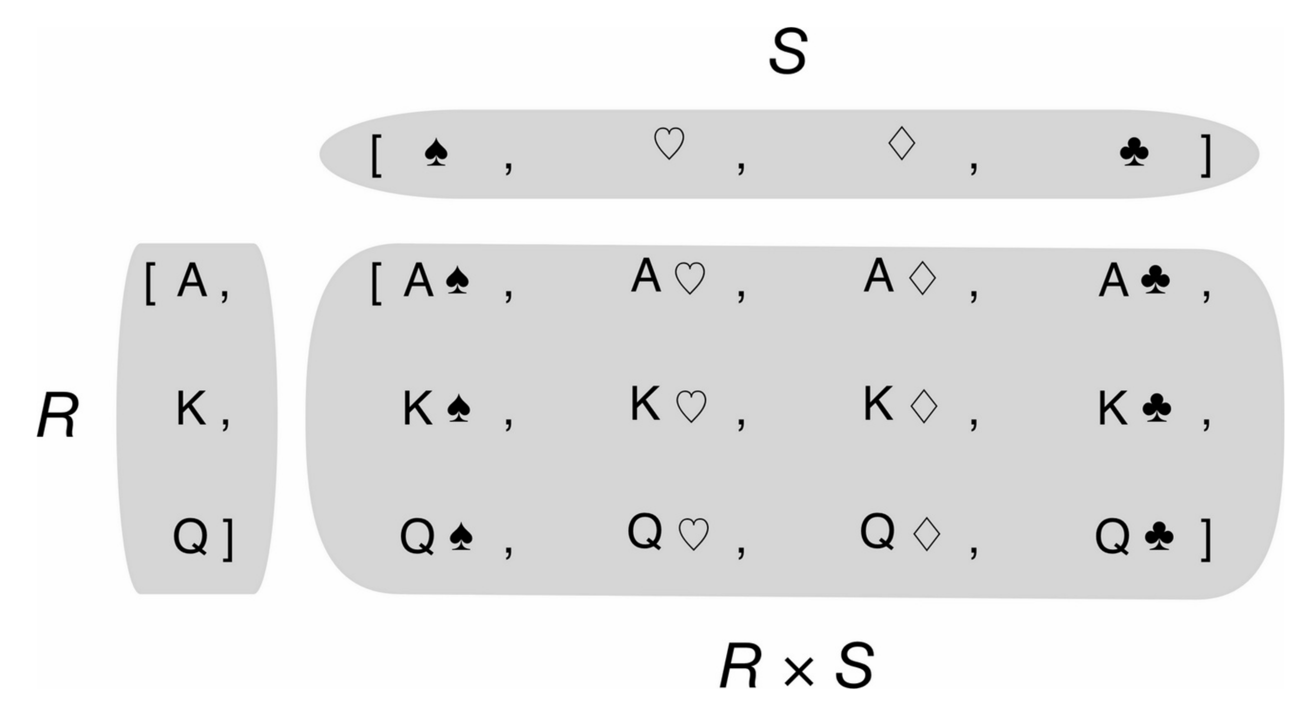


图 **2-2**：含有 **4** 种花色和 **3** 种牌面的列表的笛卡儿积，结果是一个包含 **12** 个元素的列表

如果你需要一个列表，列表里是 3 种不同尺寸的 T 恤衫，每个尺寸都有

2 个颜色，示例 2-4 用列表推导算出了这个列表，列表里有 6 种组合。

示例 **2-4**　使用列表推导计算笛卡儿积

```
>>> colors = ['black', 'white']
>>> sizes = ['S', 'M', 'L']
>>> tshirts = [(color, size) for color in colors for size in sizes] ❶
>>> tshirts
[('black', 'S'), ('black', 'M'), ('black', 'L'), ('white', 'S'),
 ('white', 'M'), ('white', 'L')]
>>> for color in colors: ❷
...     for size in sizes:
...         print((color, size))
...
('black', 'S')
('black', 'M')
('black', 'L')
('white', 'S')
('white', 'M')
('white', 'L')
>>> tshirts = [(color, size) for size in sizes    ❸
...                          for color in colors]
>>> tshirts
[('black', 'S'), ('white', 'S'), ('black', 'M'), ('white', 'M'),
 ('black', 'L'), ('white', 'L')]
```

❶ 这里得到的结果是先以颜色排列，再以尺码排列。

❷ 注意，这里两个循环的嵌套关系和上面列表推导中 for 从句的先后顺序一样。

❸ 如果想依照先尺码后颜色的顺序来排列，只需要调整从句的顺序。我在这里插入了一个换行符，这样顺序安排就更明显了。

在第 1 章的示例 1-1 中，有下面这样一段程序，它的作用是生成一个按花色分组的 52 张牌的列表，其中每个花色各有 13 张不同点数的牌。

```
self._cards = [Card(rank, suit) for suit in self.suits
                                for rank in self.ranks]
```

列表推导的作用只有一个：生成列表。如果想生成其他类型的序列，生

成器表达式就派上了用场。下一节就是对生成器表达式的一个简单介绍，其中可以看到如何用它生成列表以外的序列类型。

2.2.4　生成器表达式

虽然也可以用列表推导来初始化元组、数组或其他序列类型，但是生成器表达式是更好的选择。这是因为生成器表达式背后遵守了迭代器协议，可以逐个地产出元素，而不是先建立一个完整的列表，然后再把这个列表传递到某个构造函数里。前面那种方式显然能够节省内存。

生成器表达式的语法跟列表推导差不多，只不过把方括号换成圆括号而已。

示例 2-5 展示了如何用生成器表达式建立元组和数组。

示例 **2-5**　用生成器表达式初始化元组和数组

```
>>> symbols = '$¢£¥€¤'
>>> tuple(ord(symbol) for symbol in symbols) ❶
(36, 162, 163, 165, 8364, 164)
>>> import array
>>> array.array('I', (ord(symbol) for symbol in symbols)) ❷
array('I', [36, 162, 163, 165, 8364, 164])
```

❶ 如果生成器表达式是一个函数调用过程中的唯一参数，那么不需要额外再用括号把它围起来。

❷ `array` 的构造方法需要两个参数，因此括号是必需的。`array` 构造方法的第一个参数指定了数组中数字的存储方式。2.9.1 节中有更多关于数组的详细讨论。

示例 2-6 则是利用生成器表达式实现了一个笛卡儿积，用以打印出上文中我们提到过的 T 恤衫的 2 种颜色和 3 种尺码的所有组合。与示例 2-4 不同的是，用到生成器表达式之后，内存里不会留下一个有 6 个组合的列表，因为生成器表达式会在每次 `for` 循环运行时才生成一个组合。如果要计算两个各有 1000 个元素的列表的笛卡儿积，生成器表达式就可以帮忙省掉运行 `for` 循环的开销，即一个含有 100 万个元素的列表。

示例 **2-6**　使用生成器表达式计算笛卡儿积

```
>>> colors = ['black', 'white']
>>> sizes = ['S', 'M', 'L']
>>> for tshirt in ('%s %s' % (c, s) for c in colors for s in sizes): ❶
...     print(tshirt)
...
black S
black M
black L
white S
white M
white L
```

❶ 生成器表达式逐个产出元素，从来不会一次性产出一个含有 6 个 T 恤样式的列表。

第 14 章会专门讲到生成器的工作原理。这里只是简单看看如何用生成器来初始化除列表之外的序列，以及如何用它来避免额外的内存占用。

接下来看看 Python 中的另外一个很重要的序列类型：元组（`tuple`）。

2.3　元组不仅仅是不可变的列表

有些 Python 入门教程把元组称为“不可变列表”，然而这并没有完全概括元组的特点。除了用作不可变的列表，它还可以用于没有字段名的记录。鉴于后者常常被忽略，我们先来看看元组作为记录的功用。

2.3.1　元组和记录

元组其实是对数据的记录：元组中的每个元素都存放了记录中一个字段的数据，外加这个字段的位置。正是这个位置信息给数据赋予了意义。

如果只把元组理解为不可变的列表，那其他信息——它所含有的元素的总数和它们的位置——似乎就变得可有可无。但是如果把元组当作一些字段的集合，那么数量和位置信息就变得非常重要了。

示例 2-7 中的元组就被当作记录加以利用。如果在任何的表达式里我们在元组内对元素排序，这些元素所携带的信息就会丢失，因为这些信息是跟它们的位置有关的。

示例 **2-7**　把元组用作记录

```
>>> lax_coordinates = (33.9425, -118.408056)  ❶
>>> city, year, pop, chg, area = ('Tokyo', 2003, 32450, 0.66, 8014)  ❷
>>> traveler_ids = [('USA', '31195855'), ('BRA', 'CE342567'),  ❸
...     ('ESP', 'XDA205856')]
>>> for passport in sorted(traveler_ids):  ❹
...     print('%s/%s' % passport) ❺
...
BRA/CE342567
ESP/XDA205856
USA/31195855
>>> for country, _ in traveler_ids:  ❻
...     print(country)
...
USA
BRA
ESP
```

❶ 洛杉矶国际机场的经纬度。

❷ 东京市的一些信息：市名、年份、人口（单位：百万）、人口变化（单位：百分比）和面积（单位：平方千米）。

❸ 一个元组列表，元组的形式为 `(country_code, passport_number)`。

❹ 在迭代的过程中，`passport` 变量被绑定到每个元组上。

❺ `%` 格式运算符能被匹配到对应的元组元素上。

❻ `for` 循环可以分别提取元组里的元素，也叫作拆包（unpacking）。因为元组中第二个元素对我们没有什么用，所以它赋值给“`_`”占位符。

拆包让元组可以完美地被当作记录来使用，这也是下一节的话题。

2.3.2　元组拆包

示例 2-7 中，我们把元组 `('Tokyo', 2003, 32450, 0.66, 8014)` 里的元素分别赋值给变量 `city`、`year`、`pop`、`chg` 和 `area`，而这所有的赋值我们只用一行声明就写完了。同样，在后面一行中，一个 `%` 运算符就把 `passport` 元组里的元素对应到了 `print` 函数的格式字符串空档中。这两个都是对元组拆包的应用。

> 元组拆包可以应用到任何可迭代对象上，唯一的硬性要求是，被可迭代对象中的元素数量必须要跟接受这些元素的元组的空档数一致。除非我们用 `*` 来表示忽略多余的元素，在“用 `*` 来处理多余的元素”一节里，我会讲到它的具体用法。Python 爱好者们很喜欢用元组拆包这个说法，但是可迭代元素拆包这个表达也慢慢流行了起来，比如“PEP 3132—Extended Iterable Unpacking”（https://www.python.org/dev/peps/pep-3132/）的标题就是这么用的。

最好辨认的元组拆包形式就是平行赋值，也就是说把一个可迭代对象里的元素，一并赋值到由对应的变量组成的元组中。就像下面这段代码：

```
>>> lax_coordinates = (33.9425, -118.408056)
>>> latitude, longitude = lax_coordinates # 元组拆包
>>> latitude
33.9425
>>> longitude
-118.408056
```

另外一个很优雅的写法当属不使用中间变量交换两个变量的值：

```
>>> b, a = a, b
```

还可以用 * 运算符把一个可迭代对象拆开作为函数的参数：

```
>>> divmod(20, 8)
(2, 4)
>>> t = (20, 8)
>>> divmod(*t)
(2, 4)
>>> quotient, remainder = divmod(*t)
>>> quotient, remainder
(2, 4)
```

下面是另一个例子，这里元组拆包的用法则是让一个函数可以用元组的形式返回多个值，然后调用函数的代码就能轻松地接受这些返回值。比如 `os.path.split()` 函数就会返回以路径和最后一个文件名组成的元组 `(path, last_part)`：

```
>>> import os
>>> _, filename = os.path.split('/home/luciano/.ssh/idrsa.pub')
>>> filename
'idrsa.pub'
```

在进行拆包的时候，我们不总是对元组里所有的数据都感兴趣，`_` 占位符能帮助处理这种情况，上面这段代码也展示了它的用法。

> 如果做的是国际化软件，那么 `_` 可能就不是一个理想的占位符，因为它也是 `gettext.gettext` 函数的常用别名，`gettext` 模块的文档（https://docs.python.org/3/library/gettext.html）里提到了这一点。在其他情况下，`_` 会是一个很好的占位符。

除此之外，在元组拆包中使用 * 也可以帮助我们把注意力集中在元组的部分元素上。

用*来处理剩下的元素

在 Python 中，函数用 *args 来获取不确定数量的参数算是一种经典写法了。

于是 Python 3 里，这个概念被扩展到了平行赋值中：

```
>>> a, b, *rest = range(5)
>>> a, b, rest
(0, 1, [2, 3, 4])
>>> a, b, *rest = range(3)
>>> a, b, rest
(0, 1, [2])
>>> a, b, *rest = range(2)
>>> a, b, rest
(0, 1, [])
```

在平行赋值中，* 前缀只能用在一个变量名前面，但是这个变量可以出现在赋值表达式的任意位置：

```
>>> a, *body, c, d = range(5)
>>> a, body, c, d
(0, [1, 2], 3, 4)
>>> *head, b, c, d = range(5)
>>> head, b, c, d
([0, 1], 2, 3, 4)
```

另外元组拆包还有个强大的功能，那就是可以应用在嵌套结构中。

2.3.3　嵌套元组拆包

接受表达式的元组可以是嵌套式的，例如 (a, b, (c, d))。只要这个接受元组的嵌套结构符合表达式本身的嵌套结构，Python 就可以作出正确的对应。示例 2-8 就是对嵌套元组拆包的应用。

示例 **2-8**　用嵌套元组来获取经度

```
metro_areas = [
    ('Tokyo','JP',36.933,(35.689722,139.691667)),  # ❶
    ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),
    ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),
    ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),
    ('Sao Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
]

print('{:15} | {:^9} | {:^9}'.format('', 'lat.', 'long.'))
fmt = '{:15} | {:9.4f} | {:9.4f}'
for name, cc, pop, (latitude, longitude) in metro_areas:  # ❷
    if longitude <= 0:  # ❸
        print(fmt.format(name, latitude, longitude))
```

❶ 每个元组内有 4 个元素，其中最后一个元素是一对坐标。

❷ 我们把输入元组的最后一个元素拆包到由变量构成的元组里，这样就获取了坐标。

❸ `if longitude <= 0:` 这个条件判断把输出限制在西半球的城市。

示例 2-8 的输出是这样的：

```
                |   lat.    |   long.
Mexico City     |   19.4333 |  -99.1333
New York-Newark |   40.8086 |  -74.0204
Sao Paul        |  -23.5478 |  -46.6358
```

在 Python 3 之前，元组可以作为形参放在函数声明中，例如 `def fn(a, (b, c), d):`。然而 Python 3 不再支持这种格式，具体原因见于“PEP 3113—Removal of Tuple Parameter Unpacking”（http://python.org/dev/peps/pep-3113/）。需要弄清楚的是，这个改变对函数调用者并没有影响，它改变的是某些函数的声明方式。

元组已经设计得很好用了，但作为记录来用的话，还是少了一个功能：我们时常会需要给记录中的字段命名。`namedtuple` 函数的出现帮我们解决了这个问题。

2.3.4　具名元组

`collections.namedtuple` 是一个工厂函数，它可以用来构建一个带字段名的元组和一个有名字的类——这个带名字的类对调试程序有很大帮助。

> 用 `namedtuple` 构建的类的实例所消耗的内存跟元组是一样的，因为字段名都被存在对应的类里面。这个实例跟普通的对象实例比起来也要小一些，因为 Python 不会用 `__dict__` 来存放这些实例的属性。

在第 1 章的示例 1-1 中是这样新建 `Card` 类的：

```
Card = collections.namedtuple('Card', ['rank', 'suit'])
```

示例 2-9 展示了如何用具名元组来记录一个城市的信息。

示例 **2-9**　定义和使用具名元组

```
>>> from collections import namedtuple
>>> City = namedtuple('City', 'name country population coordinates')  ❶
>>> tokyo = City('Tokyo', 'JP', 36.933, (35.689722, 139.691667))  ❷
>>> tokyo
City(name='Tokyo', country='JP', population=36.933, coordinates=(35.689722,
139.691667))
>>> tokyo.population  ❸
36.933
>>> tokyo.coordinates
(35.689722, 139.691667)
>>> tokyo[1]
'JP'
```

❶ 创建一个具名元组需要两个参数，一个是类名，另一个是类的各个字段的名字。后者可以是由数个字符串组成的可迭代对象，或者是由空格分隔开的字段名组成的字符串。

❷ 存放在对应字段里的数据要以一串参数的形式传入到构造函数中（注意，元组的构造函数却只接受单一的可迭代对象）。

❸ 你可以通过字段名或者位置来获取一个字段的信息。

除了从普通元组那里继承来的属性之外，具名元组还有一些自己专有的属性。示例 2-10 中就展示了几个最有用的：`_fields` 类属性、类方法 `_make(iterable)` 和实例方法 `_asdict()`。

示例 **2-10**　具名元组的属性和方法（接续前一个示例）

```
>>> City._fields  ❶
('name', 'country', 'population', 'coordinates')
>>> LatLong = namedtuple('LatLong', 'lat long')
>>> delhi_data = ('Delhi NCR', 'IN', 21.935, LatLong(28.613889, 77.208889))
>>> delhi = City._make(delhi_data)  ❷
>>> delhi._asdict()  ❸
OrderedDict([('name', 'Delhi NCR'), ('country', 'IN'), ('population',
21.935), ('coordinates', LatLong(lat=28.613889, long=77.208889))])
>>> for key, value in delhi._asdict().items():
        print(key + ':', value)

name: Delhi NCR
country: IN
population: 21.935
coordinates: LatLong(lat=28.613889, long=77.208889)
>>>
```

❶ `_fields` 属性是一个包含这个类所有字段名称的元组。

❷ 用 `_make()` 通过接受一个可迭代对象来生成这个类的一个实例，它的作用跟 `City(*delhi_data)` 是一样的。

❸ `_asdict()` 把具名元组以 `collections.OrderedDict` 的形式返回，我们可以利用它来把元组里的信息友好地呈现出来。

现在我们知道了，元组是一种很强大的可以当作记录来用的数据类型。它的第二个角色则是充当一个不可变的列表。下面就来看看它的第二重功能。

2.3.5　作为不可变列表的元组

如果要把元组当作列表来用的话，最好先了解一下它们的相似度如何。

在表 2-1 中可以清楚地看到，除了跟增减元素相关的方法之外，元组支持列表的其他所有方法。还有一个例外，元组没有 `__reversed__` 方法，但是这个方法只是个优化而已，`reversed(my_tuple)` 这个用法在没有 `__reversed__` 的情况下也是合法的。

表**2-1**：列表或元组的方法和属性（那些由**`object`**类支持的方法没有列出来）

	列表	元组	
`s.__add__(s2)`	•	•	`s + s2`，拼接
`s.__iadd__(s2)`	•		`s += s2`，就地拼接
`s.append(e)`	•		在尾部添加一个新元素
`s.clear()`	•		删除所有元素
`s.__contains__(e)`	•	•	`s` 是否包含 `e`
`s.copy()`	•		列表的浅复制
`s.count(e)`	•	•	`e` 在 `s` 中出现的次数
`s.__delitem__(p)`	•		把位于 `p` 的元素删除
`s.extend(it)`	•		把可迭代对象 `it` 追加给 `s`
`s.__getitem__(p)`	•	•	`s[p]`，获取位置 `p` 的元素

s.__getnewargs__()		•	在 pickle 中支持更加优化的序列化
s.index(e)	•	•	在 s 中找到元素 e 第一次出现的位置
s.insert(p, e)	•		在位置 p 之前插入元素e
s.__iter__()	•	•	获取 s 的迭代器
s.__len__()	•	•	len(s)，元素的数量
s.__mul__(n)	•	•	s * n，n 个 s 的重复拼接
s.__imul__(n)	•		s *= n，就地重复拼接
s.__rmul__(n)	•	•	n * s，反向拼接 *
s.pop([p])	•		删除最后或者是（可选的）位于 p 的元素，并返回它的值
s.remove(e)	•		删除 s 中的第一次出现的 e
s.reverse()	•		就地把 s 的元素倒序排列
s.__reversed__()	•		返回 s 的倒序迭代器
s.__setitem__(p, e)	•		s[p] = e，把元素 e 放在位置p，替代已经在那个位置的元素
s.sort([key], [reverse])	•		就地对 s 中的元素进行排序，可选的参数有键（key）和是否倒序（reverse）

* 反向运算符在第 13 章中介绍。

每个 Python 程序员都知道序列可以用 `s[a:b]` 的形式切片，但是关于切片，我还想说说它的一些不太为人所知的方面。

2.4　切片

在 Python 里，像列表（`list`）、元组（`tuple`）和字符串（`str`）这类序列类型都支持切片操作，但是实际上切片操作比人们所想象的要强大很多。

这一节主要讨论的是这些高级切片形式的用法，它们的实现方法则会在第 10 章的一个自定义类里提到。这么做主要是为了符合这本书的哲学：先讲用法，第四部分中再来讲如何创建新类。

2.4.1　为什么切片和区间会忽略最后一个元素

在切片和区间操作里不包含区间范围的最后一个元素是 Python 的风格，这个习惯符合 Python、C 和其他语言里以 0 作为起始下标的传统。这样做带来的好处如下。

- 当只有最后一个位置信息时，我们也可以快速看出切片和区间里有几个元素：`range(3)` 和 `my_list[:3]` 都返回 3 个元素。
- 当起止位置信息都可见时，我们可以快速计算出切片和区间的长度，用后一个数减去第一个下标（`stop - start`）即可。
- 这样做也让我们可以利用任意一个下标来把序列分割成不重叠的两部分，只要写成 `my_list[:x]` 和 `my_list[x:]` 就可以了，如下所示。

```
>>> l = [10, 20, 30, 40, 50, 60]
>>> l[:2] # 在下标2的地方分割
[10, 20]
>>> l[2:]
[30, 40, 50, 60]
>>> l[:3] # 在下标3的地方分割
[10, 20, 30]
 >>> l[3:]
[40, 50, 60]
```

计算机科学家 Edsger W. Dijkstar 对这一风格的解释应该是最好的，详

见“延伸阅读”中给出的最后一个参考资料。

接下来进一步看看 Python 解释器是如何理解切片操作的。

2.4.2　对对象进行切片

一个众所周知的秘密是，我们还可以用 `s[a:b:c]` 的形式对 `s` 在 `a` 和 `b` 之间以 `c` 为间隔取值。`c` 的值还可以为负，负值意味着反向取值。下面的 3 个例子更直观些：

```
>>> s = 'bicycle'
>>> s[::3]
'bye'
>>> s[::-1]
'elcycib'
>>> s[::-2]
'eccb'
```

另一个例子是在第 1 章中用 `deck[12::13]` 的形式在未洗过的牌里把每种花色的 A 拿出来：

```
>>> deck[12::13]
[Card(rank='A', suit='spades'), Card(rank='A', suit='diamonds'),
Card(rank='A', suit='clubs'), Card(rank='A', suit='hearts')]
```

`a:b:c` 这种用法只能作为索引或者下标用在 `[]` 中来返回一个切片对象：`slice(a, b, c)`。在 10.4.1 节中会讲到，对 `seq[start:stop:step]` 进行求值的时候，Python 会调用 `seq.__getitem__(slice(start, stop, step))`。就算你还不会自定义序列类型，了解一下切片对象也是有好处的。例如你可以给切片命名，就像电子表格软件里给单元格区域取名字一样。

比如，要解析示例 2-11 中所示的纯文本文件，这时使用有名字的切片比用硬编码的数字区间要方便得多，注意示例里的 `for` 循环的可读性有多强。

示例 **2-11**　纯文本文件形式的收据以一行字符串的形式被解析

```
>>> invoice = """
```

```
... 0.....6.................................40........52...55........
... 1909  Pimoroni PiBrella                     $17.50    3    $52.50
... 1489  6mm Tactile Switch x20                 $4.95    2     $9.90
... 1510  Panavise Jr. - PV-201                 $28.00    1    $28.00
... 1601  PiTFT Mini Kit 320x240                $34.95    1    $34.95
... """
>>> SKU = slice(0, 6)
>>> DESCRIPTION = slice(6, 40)
>>> UNIT_PRICE = slice(40, 52)
>>> QUANTITY = slice(52, 55)
>>> ITEM_TOTAL = slice(55, None)
>>> line_items = invoice.split('\n')[2:]
>>> for item in line_items:
...     print(item[UNIT_PRICE], item[DESCRIPTION])
...
    $17.50   Pimoroni PiBrella
     $4.95   6mm Tactile Switch x20
    $28.00   Panavise Jr. - PV-201
    $34.95   PiTFT Mini Kit 320x240
```

在 10.4 节还有更多机会来了解切片（`slice`）对象。如果从 Python 用户的角度出发，切片还有个两个额外的功能：多维切片和省略表示法（`...`）。

2.4.3 多维切片和省略

`[]` 运算符里还可以使用以逗号分开的多个索引或者是切片，外部库 NumPy 里就用到了这个特性，二维的 `numpy.ndarray` 就可以用 `a[i, j]` 这种形式来获取，抑或是用 `a[m:n, k:l]` 的方式来得到二维切片。稍后的示例 2-22 会展示这个用法。要正确处理这种 `[]` 运算符的话，对象的特殊方法 `__getitem__` 和 `__setitem__` 需要以元组的形式来接收 `a[i, j]` 中的索引。也就是说，如果要得到 `a[i, j]` 的值，Python 会调用 `a.__getitem__((i, j))`。

Python 内置的序列类型都是一维的，因此它们只支持单一的索引，成对出现的索引是没有用的。

省略（ellipsis）的正确书写方法是三个英语句号（`...`），而不是 Unicdoe 码位 U+2026 表示的半个省略号（...）。省略在 Python 解析器眼里是一个符号，而实际上它是 Ellipsis 对象的别名，而 `Ellipsis` 对象

又是 `ellipsis` 类的单一实例。[2] 它可以当作切片规范的一部分，也可以用在函数的参数清单中，比如 `f(a, ..., z)`，或 `a[i:...]`。在 NumPy 中，`...` 用作多维数组切片的快捷方式。如果 `x` 是四维数组，那么 `x[i, ...]` 就是 `x[i, :, :, :]` 的缩写。如果想了解更多，请参见“Tentative NumPy Tutorial”（http://wiki.scipy.org/Tentative_NumPy_Tutorial）。

> [2]是的，你没看错，`ellipsis` 是类名，全小写，而它的内置实例写作 `Ellipsis`。这其实跟 `bool` 是小写，但是它的两个实例写作 `True` 和 `False` 异曲同工。

在写这本书的时候，我还没有发现在 Python 的标准库里有任何 `Ellipsis` 或者是多维索引的用法。如果你知道，请告诉我。这些句法上的特性主要是为了支持用户自定义类或者扩展，比如 NumPy 就是个例子。

除了用来提取序列里的内容，切片还可以用来就地修改可变序列，也就是说修改的时候不需要重新组建序列。

2.4.4　给切片赋值

如果把切片放在赋值语句的左边，或把它作为 `del` 操作的对象，我们就可以对序列进行嫁接、切除或就地修改操作。通过下面这几个例子，你应该就能体会到这些操作的强大功能：

```
>>> l = list(range(10))
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[2:5] = [20, 30]
>>> l
[0, 1, 20, 30, 5, 6, 7, 8, 9]
>>> del l[5:7]
>>> l
[0, 1, 20, 30, 5, 8, 9]
>>> l[3::2] = [11, 22]
>>> l
[0, 1, 20, 11, 5, 22, 9]
>>> l[2:5] = 100  ❶
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only assign an iterable
```

```
>>> l[2:5] = [100]
>>> l
[0, 1, 100, 22, 9]
```

❶ 如果赋值的对象是一个切片，那么赋值语句的右侧必须是个可迭代对象。即便只有单独一个值，也要把它转换成可迭代的序列。

序列的拼接操作可谓是众所周知，任何一本 Python 入门教材都会介绍 + 和 * 的用法，但是在这些用法的背后还有一些可能被忽视的细节。下面就来看看这两种操作。

2.5　对序列使用+和*

Python 程序员会默认序列是支持 + 和 * 操作的。通常 + 号两侧的序列由相同类型的数据所构成，在拼接的过程中，两个被操作的序列都不会被修改，Python 会新建一个包含同样类型数据的序列来作为拼接的结果。

如果想要把一个序列复制几份然后再拼接起来，更快捷的做法是把这个序列乘以一个整数。同样，这个操作会产生一个新序列：

```
>>> l = [1, 2, 3]
>>> l * 5
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> 5 * 'abcd'
'abcdabcdabcdabcdabcd'
```

+ 和 * 都遵循这个规律，不修改原有的操作对象，而是构建一个全新的序列。

> 如果在 a * n 这个语句中，序列 a 里的元素是对其他可变对象的引用的话，你就需要格外注意了，因为这个式子的结果可能会出乎意料。比如，你想用 my_list = [[]] * 3 来初始化一个由列表组成的列表，但是你得到的列表里包含的 3 个元素其实是 3 个引用，而且这 3 个引用指向的都是同一个列表。这可能不是你想要的效果。

下面来看看如何用 * 来初始化一个由列表组成的列表。

建立由列表组成的列表

有时我们会需要初始化一个嵌套着几个列表的列表，譬如一个列表可能需要用来存放不同的学生名单，或者是一个井字游戏板[3]上的一行方块。想要达成这些目的，最好的选择是使用列表推导，见示例 2-12。

[3]又称过三关，是一种在 3×3 的方块矩阵上进行的游戏。——译者注

示例 **2-12**　一个包含 3 个列表的列表，嵌套的 3 个列表各自有 3 个元素来代表井字游戏的一行方块

```
>>> board = [['_'] * 3 for i in range(3)]  ❶
>>> board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> board[1][2] = 'X' ❷
>>> board
[['_', '_', '_'], ['_', '_', 'X'], ['_', '_', '_']]
```

❶ 建立一个包含 3 个列表的列表，被包含的 3 个列表各自有 3 个元素。打印出这个嵌套列表。

❷ 把第 1 行第 2 列的元素标记为 X，再打印出这个列表。

示例 2-13 展示了另一个方法，这个方法看上去是个诱人的捷径，但实际上它是错的。

示例 **2-13**　含有 3 个指向同一对象的引用的列表是毫无用处的

```
>>> weird_board = [['_'] * 3] * 3 ❶
>>> weird_board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> weird_board[1][2] = 'O' ❷
>>> weird_board
[['_', '_', 'O'], ['_', '_', 'O'], ['_', '_', 'O']]
```

❶ 外面的列表其实包含 3 个指向同一个列表的引用。当我们不做修改的时候，看起来都还好。

❷ 一旦我们试图标记第 1 行第 2 列的元素，就立马暴露了列表内的 3 个引用指向同一个对象的事实。

示例 2-13 犯的错误本质上跟下面的代码犯的错误一样：

```
row=['_'] * 3
board = []
for i in range(3):
    board.append(row)  ❶
```

❶ 追加同一个行对象（row）3 次到游戏板（board）。

相反，示例 2-12 中的方法等同于这样做：

```
>>> board = []
>>> for i in range(3):
...     row=['_'] * 3 # ❶
...     board.append(row)
...
>>> board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> board[2][0] = 'X'
>>> board  # ❷
[['_', '_', '_'], ['_', '_', '_'], ['X', '_', '_']]
```

❶ 每次迭代中都新建了一个列表，作为新的一行（row）追加到游戏板（board）。

❷ 正如我们所期待的，只有第 2 行的元素被修改。

> 如果你觉得这一节里所说的问题及其对应的解决方法都有点云里雾里，没关系。第 8 章里我们会详细说明引用和可变对象背后的原理和陷阱。

我们一直在说 + 和 *，但是别忘了我们还有 += 和 *=。随着目标序列的可变性的变化，这个两个运算符的结果也大相径庭。下一节就来详细讨论。

2.6　序列的增量赋值

增量赋值运算符 += 和 *= 的表现取决于它们的第一个操作对象。简单起见，我们把讨论集中在增量加法（+=）上，但是这些概念对 *= 和其他增量运算符来说都是一样的。

+= 背后的特殊方法是 __iadd__ （用于"就地加法"）。但是如果一个类没有实现这个方法的话，Python 会退一步调用 __add__ 。考虑下面这个简单的表达式：

```
>>> a += b
```

如果 a 实现了 __iadd__ 方法，就会调用这个方法。同时对可变序列（例如 list、bytearray 和 array.array）来说，a 会就地改动，就像调用了 a.extend(b) 一样。但是如果 a 没有实现 __iadd__ 的话，a += b 这个表达式的效果就变得跟 a = a + b 一样了：首先计算 a + b，得到一个新的对象，然后赋值给 a。也就是说，在这个表达式中，变量名会不会被关联到新的对象，完全取决于这个类型有没有实现 __iadd__ 这个方法。

总体来讲，可变序列一般都实现了 __iadd__ 方法，因此 += 是就地加法。而不可变序列根本就不支持这个操作，对这个方法的实现也就无从谈起。

上面所说的这些关于 += 的概念也适用于 *=，不同的是，后者相对应的是 __imul__。关于 __iadd__ 和 __imul__，第 13 章中会再次提到。

接下来有个小例子，展示的是 *= 在可变和不可变序列上的作用：

```
>>> l = [1, 2, 3]
>>> id(l)
4311953800  ❶
>>> l *= 2
>>> l
[1, 2, 3, 1, 2, 3]
>>> id(l)
4311953800  ❷
```

```
>>> t = (1, 2, 3)
>>> id(t)
4312681568  ❸
>>> t *= 2
>>> id(t)
4301348296  ❹
```

❶ 刚开始时列表的 ID。

❷ 运用增量乘法后，列表的 ID 没变，新元素追加到列表上。

❸ 元组最开始的 ID。

❹ 运用增量乘法后，新的元组被创建。

对不可变序列进行重复拼接操作的话，效率会很低，因为每次都有一个新对象，而解释器需要把原来对象中的元素先复制到新的对象里，然后再追加新的元素。[4]

> [4]`str` 是一个例外，因为对字符串做 `+=` 实在是太普遍了，所以 CPython 对它做了优化。为 `str` 初始化内存的时候，程序会为它留出额外的可扩展空间，因此进行增量操作的时候，并不会涉及复制原有字符串到新位置这类操作。

我们已经认识了 `+=` 的一般用法，下面来看一个有意思的边界情况。这个例子可以说是突出展示了“不可变性”对于元组来说到底意味着什么。

一个关于+=的谜题

读完下面的代码，然后回答这个问题：示例 2-14 中的两个表达式到底会产生什么结果？[5]回答之前不要用控制台去运行这两个式子。

> [5]感谢 Leonardo Rochael 在 2013 年的 Python 巴西会议上提到这个谜题。

示例 **2-14**　一个谜题

```
>>> t = (1, 2, [30, 40])
>>> t[2] += [50, 60]
```

到底会发生下面 4 种情况中的哪一种？

a. `t` 变成 `(1, 2, [30, 40, 50, 60])`。

b. 因为 `tuple` 不支持对它的元素赋值，所以会抛出 `TypeError` 异常。

c. 以上两个都不是。

d. **a** 和 **b** 都是对的。

我刚看到这个问题的时候，异常确定地选择了 **b**，但其实答案是 **d**，也就是说 **a** 和 **b** 都是对的！示例 2-15 是运行这段代码得到的结果，用的 Python 版本是 3.4，但是在 2.7 中结果也一样。[6]

> [6]有读者提出，如果写成 `t[2].extend([50, 60])` 就能避免这个异常。确实是这样，但这个例子是为了展示这种奇怪的现象而专门写的。

示例 **2-15**　没人料到的结果：`t[2]` 被改动了，但是也有异常抛出

```
>>> t = (1, 2, [30, 40])
>>> t[2] += [50, 60]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> t
(1, 2, [30, 40, 50, 60])
```

Python Tutor（http://www.pythontutor.com）是一个对 Python 运行原理进行可视化分析的工具。图 2-3 里是两张截图，分别代表示例 2-15 中 `t` 的初始和最终状态。

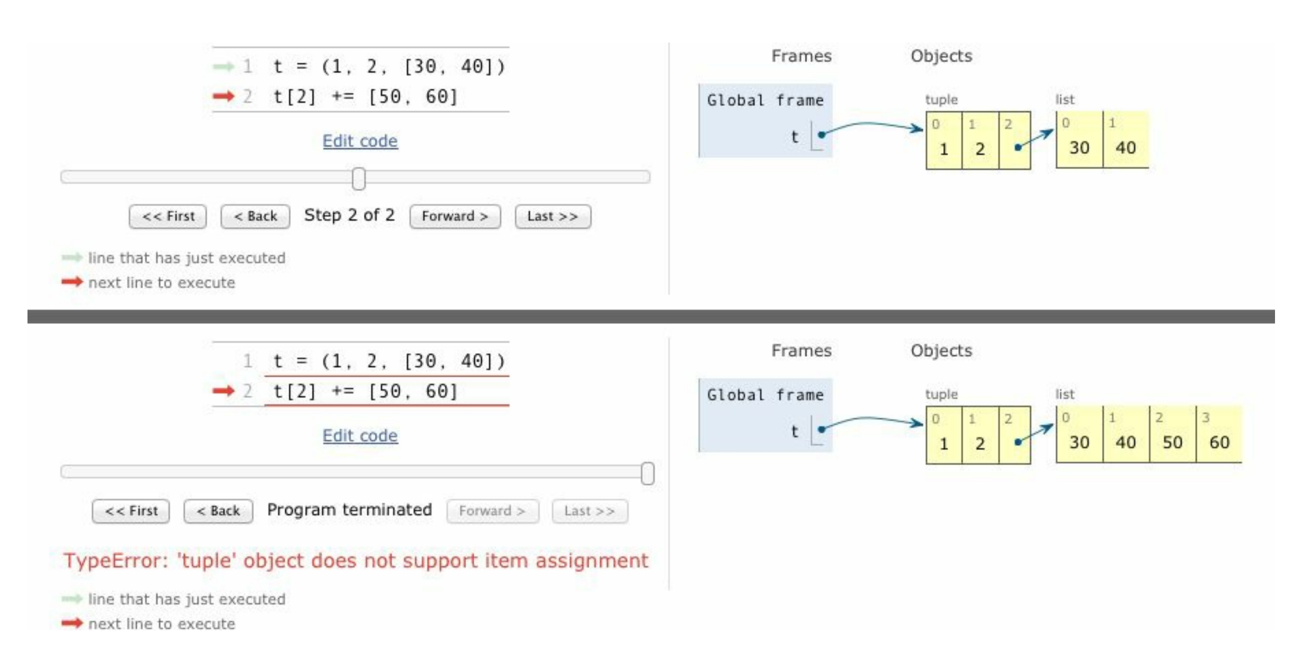


图 **2-3**：元组赋值之谜的初始和最终状态（图表由 **Python Tutor** 网站生成）

下面来看看示例 2-16 中 Python 为表达式 `s[a] += b` 生成的字节码，可能这个现象背后的原因会变得清晰起来。

示例 **2-16**　`s[a] = b` 背后的字节码

```
>>> dis.dis('s[a] += b')
  1           0 LOAD_NAME                0(s)
              3 LOAD_NAME                1(a)
              6 DUP_TOP_TWO
              7 BINARY_SUBSCR                ❶
              8 LOAD_NAME                2(b)
             11 INPLACE_ADD                  ❷
             12 ROT_THREE
             13 STORE_SUBSCR                 ❸
             14 LOAD_CONST               0(None)
             17 RETURN_VALUE
```

❶ 将 `s[a]` 的值存入 `TOS`（Top Of Stack，栈的顶端）。

❷ 计算 `TOS += b`。这一步能够完成，是因为 `TOS` 指向的是一个可变对象（也就是示例 2-15 里的列表）。

❸ `s[a] = TOS` 赋值。这一步失败，是因为 `s` 是不可变的元组（示例 2-

15 中的元组 `t`）。

这其实是个非常罕见的边界情况，在 15 年的 Python 生涯中，我还没见过谁在这个地方吃过亏。

至此我得到了 3 个教训。

- 不要把可变对象放在元组里面。
- 增量赋值不是一个原子操作。我们刚才也看到了，它虽然抛出了异常，但还是完成了操作。
- 查看 Python 的字节码并不难，而且它对我们了解代码背后的运行机制很有帮助。

在见证了 `+` 和 `*` 的微妙之处后，我们把话题转移到序列类型的另一个重要部分上：排序。

2.7　list.sort方法和内置函数sorted

list.sort 方法会就地排序列表，也就是说不会把原列表复制一份。这也是这个方法的返回值是 None 的原因，提醒你本方法不会新建一个列表。在这种情况下返回 None 其实是 Python 的一个惯例：如果一个函数或者方法对对象进行的是就地改动，那它就应该返回 None，好让调用者知道传入的参数发生了变动，而且并未产生新的对象。例如，random.shuffle 函数也遵守了这个惯例。

> 用返回 None 来表示就地改动这个惯例有个弊端，那就是调用者无法将其串联起来。而返回一个新对象的方法（比如说 str 里的所有方法）则正好相反，它们可以串联起来调用，从而形成连贯接口（fluent interface）。详情参见维基百科中有关连贯接口的讨论（https://en.wikipedia.org/wiki/Fluent_interface）。

与 list.sort 相反的是内置函数 sorted，它会新建一个列表作为返回值。这个方法可以接受任何形式的可迭代对象作为参数，甚至包括不可变序列或生成器（见第 14 章）。而不管 sorted 接受的是怎样的参数，它最后都会返回一个列表。

不管是 list.sort 方法还是 sorted 函数，都有两个可选的关键字参数。

reverse

　　如果被设定为 True，被排序的序列里的元素会以降序输出（也就是说把最大值当作最小值来排序）。这个参数的默认值是 False。

key

　　一个只有一个参数的函数，这个函数会被用在序列里的每一个元素上，所产生的结果将是排序算法依赖的对比关键字。比如说，在对一些字符串排序时，可以用 key=str.lower 来实现忽略大小写的排序，或者是用 key=len 进行基于字符串长度的排序。这个参数的默认值是恒

等函数（identity function），也就是默认用元素自己的值来排序。

可选参数 `key` 还可以在内置函数 `min()` 和 `max()` 中起作用。另外，还有些标准库里的函数也接受这个参数，像 `itertools.groupby()` 和 `heapq.nlargest()` 等。

下面通过几个小例子来看看这两个函数和它们的关键字参数：[7]

> [7]这几个例子还说明了 Python 的排序算法——Timsort——是稳定的，意思是就算两个元素比不出大小，在每次排序的结果里它们的相对位置是固定的。Timsort 在本章结尾的"杂谈"里会有进一步的讨论。

```
>>> fruits = ['grape', 'raspberry', 'apple', 'banana']
>>> sorted(fruits)
['apple', 'banana', 'grape', 'raspberry']  ❶
>>> fruits
['grape', 'raspberry', 'apple', 'banana']  ❷
>>> sorted(fruits, reverse=True)
['raspberry', 'grape', 'banana', 'apple']  ❸
>>> sorted(fruits, key=len)
['grape', 'apple', 'banana', 'raspberry']  ❹
>>> sorted(fruits, key=len, reverse=True)
['raspberry', 'banana', 'grape', 'apple']  ❺
>>> fruits
['grape', 'raspberry', 'apple', 'banana']  ❻
>>> fruits.sort()                          ❼
>>> fruits
['apple', 'banana', 'grape', 'raspberry']  ❽
```

❶ 新建了一个按照字母排序的字符串列表。

❷ 原列表并没有变化。

❸ 按照字母降序排序。

❹ 新建一个按照长度排序的字符串列表。因为这个排序算法是稳定的，grape 和 apple 的长度都是 5，它们的相对位置跟在原来的列表里是一样的。

❺ 按照长度降序排序的结果。结果并不是上面那个结果的完全翻转，

因为用到的排序算法是稳定的，也就是说在长度一样时，grape 和 apple 的相对位置不会改变。

❻ 直到这一步，原列表 `fruits` 都没有任何变化。

❼ 对原列表就地排序，返回值 None 会被控制台忽略。

❽ 此时 `fruits` 本身被排序。

已排序的序列可以用来进行快速搜索，而标准库的 `bisect` 模块给我们提供了二分查找算法。下一节会详细讲这个函数，顺便还会看看 `bisect.insort` 如何让已排序的序列保持有序。

2.8 用bisect来管理已排序的序列

bisect 模块包含两个主要函数，bisect 和 insort，两个函数都利用二分查找算法来在有序序列中查找或插入元素。

2.8.1 用bisect来搜索

bisect(haystack, needle) 在 haystack（干草垛）里搜索 needle（针）的位置，该位置满足的条件是，把 needle 插入这个位置之后，haystack 还能保持升序。也就是在说这个函数返回的位置前面的值，都小于或等于 needle 的值。其中 haystack 必须是一个有序的序列。你可以先用 bisect(haystack, needle) 查找位置 index，再用 haystack.insert(index, needle) 来插入新值。但你也可用 insort 来一步到位，并且后者的速度更快一些。

Python 的高产贡献者 Raymond Hettinger 写了一个排序集合模块（http://code.activestate.com/recipes/577197-sortedcollection/），模块里集成了 bisect 功能，但是比独立的 bisect 更易用。

示例 2-17 利用几个精心挑选的 needle，向我们展示了 bisect 返回的不同位置值。这段代码的输出结果显示在图 2-4 中。

示例 **2-17** 在有序序列中用 bisect 查找某个元素的插入位置

```
import bisect
import sys

HAYSTACK = [1, 4, 5, 6, 8, 12, 15, 20, 21, 23, 23, 26, 29, 30]
NEEDLES = [0, 1, 2, 5, 8, 10, 22, 23, 29, 30, 31]

ROW_FMT = '{0:2d} @ {1:2d}    {2}{0:<2d}'

def demo(bisect_fn):
    for needle in reversed(NEEDLES):
        position = bisect_fn(HAYSTACK, needle)  ❶
        offset = position * '  |'  ❷
```

```
        print(ROW_FMT.format(needle, position, offset))  ❸

if __name__ == '__main__':

    if sys.argv[-1] == 'left':  ❹
        bisect_fn = bisect.bisect_left
    else:
        bisect_fn = bisect.bisect

    print('DEMO:', bisect_fn.__name__)  ❺
    print('haystack ->', ' '.join('%2d' % n for n in HAYSTACK))
    demo(bisect_fn)
```

❶ 用特定的 bisect 函数来计算元素应该出现的位置。

❷利用该位置来算出需要几个分隔符号。

❸ 把元素和其应该出现的位置打印出来。

❹ 根据命令上最后一个参数来选用 bisect 函数。

❺ 把选定的函数在抬头打印出来。

```
02-array-seq/ $ python3 bisect_demo.py
DEMO: bisect
haystack ->  1  4  5  6  8 12 15 20 21 23 23 26 29 30
31 @ 14      |  |  |  |  |  |  |  |  |  |  |  |  |  |31
30 @ 14      |  |  |  |  |  |  |  |  |  |  |  |  |  |30
29 @ 13      |  |  |  |  |  |  |  |  |  |  |  |  |29
23 @ 11      |  |  |  |  |  |  |  |  |  |  |23
22 @  9      |  |  |  |  |  |  |  |  |22
10 @  5      |  |  |  |  |10
 8 @  5      |  |  |  |  |8
 5 @  3      |  |  |5
 2 @  1      |2
 1 @  1      |1
 0 @  0    0
```

图 **2-4**：用 **bisect** 函数时示例 **2-17** 的输出。每一行以 **needle @ position**（元素及其应该插入的位置）开始，然后展示了该元素在原序列中的物理位置

bisect 的表现可以从两个方面来调教。

首先可以用它的两个可选参数——lo 和 hi——来缩小搜寻的范围。lo 的默认值是 0，hi 的默认值是序列的长度，即 len() 作用于该序列的返回值。

其次，bisect 函数其实是 bisect_right 函数的别名，后者还有个姊妹函数叫 bisect_left。它们的区别在于，bisect_left 返回的插入位置是原序列中跟被插入元素相等的元素的位置，也就是新元素会被放置于它相等的元素的前面，而 bisect_right 返回的则是跟它相等的元素之后的位置。这个细微的差别可能对于整数序列来讲没什么用，但是对于那些值相等但是形式不同的数据类型来讲，结果就不一样了。比如说虽然 1 == 1.0 的返回值是 True，1 和 1.0 其实是两个不同的元素。图 2-5 显示的是用 bisect_left 来运行上述示例的结果。

```
02-array-seq/ $ python3 bisect_demo.py left
DEMO: bisect_left
haystack ->  1  4  5  6  8 12 15 20 21 23 23 26 29 30
31 @ 14      |  |  |  |  |  |  |  |  |  |  |  |  |  |31
30 @ 13      |  |  |  |  |  |  |  |  |  |  |  |  |30
29 @ 12      |  |  |  |  |  |  |  |  |  |  |  |29
23 @  9      |  |  |  |  |  |  |  |  |23
22 @  9      |  |  |  |  |  |  |  |  |22
10 @  5      |  |  |  |  |10
 8 @  4      |  |  |  |8
 5 @  2      |  |5
 2 @  1      |2
 1 @  0    1
 0 @  0    0
```

图 **2-5**：用 **bisect_left** 运行示例 **2-17** 得到的结果（跟图 **2-4** 对比可以发现，值 **1**、**8**、**23**、**29** 和 **30** 的插入位置变成了原序列中这些

值的前面）

bisect 可以用来建立一个用数字作为索引的查询表格，比如说把分数和成绩[8] 对应起来，见示例 2-18。

> [8]成绩指的是在美国大学中普遍使用的 A~F 字母成绩，A 表示优秀，F 表示不及格。——译者注

示例 **2-18**　根据一个分数，找到它所对应的成绩

```
>>> def grade(score, breakpoints=[60, 70, 80, 90], grades='FDCBA'):
...     i = bisect.bisect(breakpoints, score)
...     return grades[i]
...
>>> [grade(score) for score in [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']
```

示例 2-18 里的代码来自 bisect 模块的文档（https://docs.python.org/3/library/bisect.html）。文档里列举了一些利用 bisect 的函数，它们可以在很长的有序序列中作为 index 的替代，用来更快地查找一个元素的位置。

这些函数不但可以用于查找，还可以用来向序列中插入新元素，下面就来看看它们的用法。

2.8.2　用bisect.insort插入新元素

排序很耗时，因此在得到一个有序序列之后，我们最好能够保持它的有序。bisect.insort 就是为了这个而存在的。

insort(seq, item) 把变量 item 插入到序列 seq 中，并能保持 seq 的升序顺序。详见示例 2-19 和它在图 2-6 里的输出。

示例 **2-19**　insort 可以保持有序序列的顺序

```
import bisect
import random

SIZE=7
```

```
random.seed(1729)

my_list = []
for i in range(SIZE):
    new_item = random.randrange(SIZE*2)
    bisect.insort(my_list, new_item)
    print('%2d ->' % new_item, my_list)
```

```
02-array-seq/ $ python3 bisect_insort.py
10 -> [10]
 0 -> [0, 10]
 6 -> [0, 6, 10]
 8 -> [0, 6, 8, 10]
 7 -> [0, 6, 7, 8, 10]
 2 -> [0, 2, 6, 7, 8, 10]
10 -> [0, 2, 6, 7, 8, 10, 10]
```

图 **2-6**：示例 **2-19** 的输出

`insort` 跟 `bisect` 一样，有 `lo` 和 `hi` 两个可选参数用来控制查找的范围。它也有个变体叫 `insort_left`，这个变体在背后用的是 `bisect_left`。

目前所提到的内容都不仅仅是对列表或者元组有效，还可以应用于几乎所有的序列类型上。有时候因为列表实在是太方便了，所以 Python 程序员可能会过度使用它，反正我知道我犯过这个毛病。而如果你只需要处理数字列表的话，数组可能是个更好的选择。下面就来讨论一些可以替换列表的数据结构。

2.9　当列表不是首选时

虽然列表既灵活又简单，但面对各类需求时，我们可能会有更好的选择。比如，要存放 1000 万个浮点数的话，数组（`array`）的效率要高得多，因为数组在背后存的并不是 `float` 对象，而是数字的机器翻译，也就是字节表述。这一点就跟 C 语言中的数组一样。再比如说，如果需要频繁对序列做先进先出的操作，`deque`（双端队列）的速度应该会更快。

> 如果在你的代码里，包含操作（比如检查一个元素是否出现在一个集合中）的频率很高，用 `set`（集合）会更合适。`set` 专为检查元素是否存在做过优化。但是它并不是序列，因为 `set` 是无序的。第 3 章会详细讨论它。

本章余下的内容都是关于在某些情况下可以替换列表的数据类型的，让我们从数组开始。

2.9.1　数组

如果我们需要一个只包含数字的列表，那么 `array.array` 比 `list` 更高效。数组支持所有跟可变序列有关的操作，包括 `.pop`、`.insert` 和 `.extend`。另外，数组还提供从文件读取和存入文件的更快的方法，如 `.frombytes` 和 `.tofile`。

Python 数组跟 C 语言数组一样精简。创建数组需要一个类型码，这个类型码用来表示在底层的 C 语言应该存放怎样的数据类型。比如 `b` 类型码代表的是有符号的字符（`signed char`），因此 `array('b')` 创建出的数组就只能存放一个字节大小的整数，范围从 -128 到 127，这样在序列很大的时候，我们能节省很多空间。而且 Python 不会允许你在数组里存放除指定类型之外的数据。

示例 2-20 展示了从创建一个有 1000 万个随机浮点数的数组开始，到如何把这个数组存放到文件里，再到如何从文件读取这个数组。

示例 **2-20**　一个浮点型数组的创建、存入文件和从文件读取的过程

```
>>> from array import array  ❶
>>> from random import random
>>> floats = array('d', (random() for i in range(10**7)))  ❷
>>> floats[-1]  ❸
0.07802343889111107
>>> fp = open('floats.bin', 'wb')
>>> floats.tofile(fp)  ❹
>>> fp.close()
>>> floats2 = array('d')  ❺
>>> fp = open('floats.bin', 'rb')
>>> floats2.fromfile(fp, 10**7)  ❻
>>> fp.close()
>>> floats2[-1]  ❼
0.07802343889111107
>>> floats2 == floats  ❽
True
```

❶ 引入 `array` 类型。

❷ 利用一个可迭代对象来建立一个双精度浮点数组（类型码是 `'d'`），这里我们用的可迭代对象是一个生成器表达式。

❸ 查看数组的最后一个元素。

❹ 把数组存入一个二进制文件里。

❺ 新建一个双精度浮点空数组。

❻ 把 1000 万个浮点数从二进制文件里读取出来。

❼ 查看新数组的最后一个元素。

❽ 检查两个数组的内容是不是完全一样。

从上面的代码我们能得出结论，`array.tofile` 和 `array.fromfile` 用起来很简单。把这段代码跑一跑，你还会发现它的速度也很快。一个小试验告诉我，用 `array.fromfile` 从一个二进制文件里读出 1000 万个双精度浮点数只需要 0.1 秒，这比从文本文件里读取的速度要快 60 倍，因为后者会使用内置的 `float` 方法把每一行文字转换成浮点数。

另外，使用 array.tofile 写入到二进制文件，比以每行一个浮点数的方式把所有数字写入到文本文件要快 7 倍。另外，1000 万个这样的数在二进制文件里只占用 80 000 000 个字节（每个浮点数占用 8 个字节，不需要任何额外空间），如果是文本文件的话，我们需要 181 515 739 个字节。

> 另外一个快速序列化数字类型的方法是使用 pickle（https://docs.python.org/3/library/pickle.html）模块。pickle.dump 处理浮点数组的速度几乎跟 array.tofile 一样快。不过前者可以处理几乎所有的内置数字类型，包含复数、嵌套集合，甚至用户自定义的类。前提是这些类没有什么特别复杂的实现。

还有一些特殊的数字数组，用来表示二进制数据，比如光栅图像。里面涉及的 bytes 和 bytearry 类型会在第 4 章提及。

表 2-2 对数组和列表的功能做了一些总结。

表**2-2**：列表和数组的属性和方法（不包含过期的数组方法以及那些由对象实现的方法）

	列表	数组	
s.__add(s2)__	•	•	s + s2，拼接
s.__iadd(s2)__	•	•	s += s2，就地拼接
s.append(e)	•	•	在尾部添加一个元素
s.byteswap		•	翻转数组内每个元素的字节序列，转换字节序
s.clear()	•		删除所有元素

s.__contains__(e)	•	•	s 是否含有 e
s.copy()	•		对列表浅复制
s.__copy__()		•	对 copy.copy 的支持
s.count(e)	•	•	s 中 e 出现的次数
s.__deepcopy__()		•	对 copy.deepcopy 的支持
s.__delitem__(p)	•	•	删除位置 p 的元素
s.extend(it)	•	•	将可迭代对象 it 里的元素添加到尾部
s.frombytes(b)		•	将压缩成机器值的字节序列读出来添加到尾部
s.fromfile(f, n)		•	将二进制文件 f 内含有机器值读出来添加到尾部，最多添加 n 项
s.fromlist(l)		•	将列表里的元素添加到尾部，如果其中任何一个元素导致了 TypeError 异常，那么所有的添加都会取消
s.__getitem__(p)	•	•	s[p]，读取位置 p 的元素
s.index(e)	•	•	找到 e 在序列中第一次出现的位置
s.insert(p, e)	•	•	在位于 p 的元素之前插入元素 e
s.itemsize		•	数组中每个元素的长度是几个字节
s.__iter__()			

	•	•	返回迭代器
s.__len__()	•	•	len(s)，序列的长度
s.__mul__(n)	•	•	s * n，重复拼接
s.__imul__(n)	•	•	s *= n，就地重复拼接
s.__rmul__(n)	•	•	n * s，反向重复拼接*
s.pop([p])	•	•	删除位于 p 的值并返回这个值，p 的默认值是最后一个元素的位置
s.remove(e)	•	•	删除序列里第一次出现的 e 元素
s.reverse()	•	•	就地调转序列中元素的位置
s.__reversed__()	•		返回一个从尾部开始扫描元素的迭代器
s.__setitem__(p, e)	•	•	s[p] = e，把位于 p 位置的元素替换成 e
s.sort([key], [revers])	•		就地排序序列，可选参数有 key 和 reverse
s.tobytes()		•	把所有元素的机器值用 bytes 对象的形式返回
s.tofile(f)		•	把所有元素以机器值的形式写入一个文件
s.tolist()		•	把数组转换成列表，列表里的元素类型是数字对象
			返回只有一个字符的字符串，代表数组元素在 C 语言中

s.typecode		•	的类型

* 第 13 章会讲反向运算符。



从 Python 3.4 开始，数组类型不再支持诸如 `list.sort()` 这种就地排序方法。要给数组排序的话，得用 `sorted` 函数新建一个数组：

```
a = array.array(a.typecode, sorted(a))
```

想要在不打乱次序的情况下为数组添加新的元素，`bisect.insort` 还是能派上用场（就像 2.8.2 节中所展示的）。

如果你总是跟数组打交道，却没有听过 `memoryview`，那就太遗憾了。下面就来谈谈 `memoryview`。

2.9.2　内存视图

`memoryview` 是一个内置类，它能让用户在不复制内容的情况下操作同一个数组的不同切片。`memoryview` 的概念受到了 NumPy 的启发（参见 2.9.3 节）。Travis Oliphant 是 NumPy 的主要作者，他在回答“ When should a memoryview be used?”（http://stackoverflow.com/questions/4845418/when-should-a-memoryview-be-used/）这个问题时是这样说的：

> 内存视图其实是泛化和去数学化的 NumPy 数组。它让你在不需要复制内容的前提下，在数据结构之间共享内存。其中数据结构可以是任何形式，比如 PIL 图片、SQLite 数据库和 NumPy 的数组，等等。这个功能在处理大型数据集合的时候非常重要。

`memoryview.cast` 的概念跟数组模块类似，能用不同的方式读写同一块内存数据，而且内容字节不会随意移动。这听上去又跟 C 语言中类型转换的概念差不多。`memoryview.cast` 会把同一块内存里的内容打包

成一个全新的 memoryview 对象给你。

在示例 2-21 里，我们利用 memoryview 精准地修改了一个数组的某个字节，这个数组的元素是 16 位二进制整数。

示例 **2-21** 通过改变数组中的一个字节来更新数组里某个元素的值

```
>>> numbers = array.array('h', [-2, -1, 0, 1, 2])
>>> memv = memoryview(numbers)  ❶
>>> len(memv)
5
>>> memv[0]  ❷
-2
>>> memv_oct = memv.cast('B')  ❸
>>> memv_oct.tolist()  ❹
[254, 255, 255, 255, 0, 0, 1, 0, 2, 0]
>>> memv_oct[5] = 4  ❺
>>> numbers
array('h', [-2, -1, 1024, 1, 2])  ❻
```

❶ 利用含有 5 个短整型有符号整数的数组（类型码是 'h'）创建一个 memoryview。

❷ memv 里的 5 个元素跟数组里的没有区别。

❸ 创建一个 memv_oct，这一次是把 memv 里的内容转换成 'B' 类型，也就是无符号字符。

❹ 以列表的形式查看 memv_oct 的内容。

❺ 把位于位置 5 的字节赋值成 4。

❻ 因为我们把占 2 个字节的整数的高位字节改成了 4，所以这个有符号整数的值就变成了 1024。

在第 4 章的示例 4-4 中，我们还可以看到如何利用 memoryview 和 struct 来操作二进制序列。

另外，如果利用数组来做高级的数字处理是你的日常工作，那么 NumPy 和 SciPy 应该是你的常用武器。下面就是对这两个库的简单介绍。

2.9.3 NumPy和SciPy

整本书我都在强调如何最大限度地利用 Python 标准库。但是 NumPy 和 SciPy 的优秀让我觉得偶尔跑个题来谈谈它们也是很值得的。

凭借着 NumPy 和 SciPy 提供的高阶数组和矩阵操作，Python 成为科学计算应用的主流语言。NumPy 实现了多维同质数组（homogeneous array）和矩阵，这些数据结构不但能处理数字，还能存放其他由用户定义的记录。通过 NumPy，用户能对这些数据结构里的元素进行高效的操作。

SciPy 是基于 NumPy 的另一个库，它提供了很多跟科学计算有关的算法，专为线性代数、数值积分和统计学而设计。SciPy 的高效和可靠性归功于其背后的 C 和 Fortran 代码，而这些跟计算有关的部分都源自于 Netlib 库（http://www.netlib.org）。换句话说，SciPy 把基于 C 和 Fortran 的工业级数学计算功能用交互式且高度抽象的 Python 包装起来，让科学家如鱼得水。

示例 2-22 是一个很简短的演示，从中可以窥见一些 NumPy 二维数组的基本操作。

示例 **2-22** 对 numpy.ndarray 的行和列进行基本操作

```
>>> import numpy  ❶
>>> a = numpy.arange(12)  ❷
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9,  10,  11])
>>> type(a)
<class 'numpy.ndarray'>
>>> a.shape  ❸
(12,)
>>> a.shape = 3, 4  ❹
>>> a
array([[  0,  1,  2,  3],
       [  4,  5,  6,  7],
       [  8,  9, 10, 11]])
>>> a[2]  ❺
array([  8,  9,  10,  11])
>>> a[2, 1]  ❻
9
>>> a[:, 1]  ❼
array([1, 5, 9])
>>> a.transpose() ❽
```

```
array([[  0,  4,  8],
       [  1,  5,  9],
       [  2,  6, 10],
       [  3,  7, 11]])
```

❶ 安装 NumPy 之后，导入它（NumPy 并不是 Python 标准库的一部分）。

❷ 新建一个 0~11 的整数的 `numpy.ndarry`，然后把它打印出来。

❸ 看看数组的维度，它是一个一维的、有 12 个元素的数组。

❹ 把数组变成二维的，然后把它打印出来看看。

❺ 打印出第 2 行。

❻ 打印第 2 行第 1 列的元素。

❼ 把第 1 列打印出来。

❽ 把行和列交换，就得到了一个新数组。

NumPy 也可以对 `numpy.ndarray` 中的元素进行抽象的读取、保存和其他操作：

```
>>> import numpy
>>> floats = numpy.loadtxt('floats-10M-lines.txt')  ❶
>>> floats[-3:]  ❷
array([ 3016362.69195522, 535281.10514262, 4566560.44373946])
>>> floats *= .5  ❸
>>> floats[-3:]
array([ 1508181.34597761, 267640.55257131, 2283280.22186973])
>>> from time import perf_counter as pc  ❹
>>> t0 = pc(); floats /= 3; pc() - t0  ❺
0.03690556302899495
>>> numpy.save('floats-10M', floats)  ❻
>>> floats2 = numpy.load('floats-10M.npy', 'r+')  ❼
>>> floats2 *= 6
>>> floats2[-3:]  ❽
memmap([3016362.69195522, 535281.10514262, 4566560.44373946])
```

❶ 从文本文件里读取 1000 万个浮点数。

❷ 利用序列切片来读取其中的最后 3 个数。

❸ 把数组里的每个数都乘以 0.5，然后再看看最后 3 个数。

❹ 导入精度和性能都比较高的计时器（Python 3.3 及更新的版本中都有这个库）。

❺ 把每个元素都除以 3，可以看到处理 1000 万个浮点数所需的时间还不足 40 毫秒。

❻ 把数组存入后缀为 .npy 的二进制文件。

❼ 将上面的数据导入到另外一个数组里，这次 `load` 方法利用了一种叫作内存映射的机制，它让我们在内存不足的情况下仍然可以对数组做切片。

❽ 把数组里每个数乘以 6 之后，再检视一下数组的最后 3 个数。



NumPy 和 SciPy 的安装可能会比较费劲。在“Installing the SciPy Stack”（http://www.scipy.org/install.html）页面，SciPy.org 建议找一个科学计算 Python 的分发渠道帮忙，比如 Anacoda、Enthought Canopy、WinPython，等等。常见的 GNU/Linux 版本的用户应该可以在他们自己的包管理系统中找到 NumPy 和 SciPy。例如，在 Debian 或者 Ubuntu 上面，用户可以通过下面的命令一键安装：

```
$ sudo apt-get install python-numpy python-scipy
```

以上的内容仅仅是九牛一毛。NumPy 和 SciPy 都是异常强大的库，也是其他一些很有用的工具的基石。Pandas（http://pandas.pydata.org）和 Blaze（http://blaze.pydata.org）数据分析库就以它们为基础，提供了高效的且能存储非数值类数据的数组类型，和读写常见数据文件格式（例如 csv、xls、SQL 转储和 HDF5）的功能。因此，要详细介绍 NumPy 和 SciPy 的话，不写成几本书是不可能的。虽然本书不在此列，但是如果要对 Python 的序列类型做一个概览，恐怕没有人能忽略 NumPy。

在介绍完扁平序列（包括标准数组和 NumPy 数组）之后，让我们把目光投向 Python 中可以取代列表的另外一种数据结构：队列。

2.9.4　双向队列和其他形式的队列

利用 `.append` 和 `.pop` 方法，我们可以把列表当作栈或者队列来用（比如，把 `.append` 和 `.pop(0)` 合起来用，就能模拟栈的“先进先出”的特点）。但是删除列表的第一个元素（抑或是在第一个元素之前添加一个元素）之类的操作是很耗时的，因为这些操作会牵扯到移动列表里的所有元素。

`collections.deque` 类（双向队列）是一个线程安全、可以快速从两端添加或者删除元素的数据类型。而且如果想要有一种数据类型来存放“最近用到的几个元素”，`deque` 也是一个很好的选择。这是因为在新建一个双向队列的时候，你可以指定这个队列的大小，如果这个队列满员了，还可以从反向端删除过期的元素，然后在尾端添加新的元素。示例 2-23 中有几个双向队列的典型操作。

示例 **2-23**　使用双向队列

```
>>> from collections import deque
>>> dq = deque(range(10), maxlen=10)  ❶
>>> dq
deque([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
>>> dq.rotate(3)  ❷
>>> dq
deque([7, 8, 9, 0, 1, 2, 3, 4, 5, 6], maxlen=10)
>>> dq.rotate(-4)
>>> dq
deque([1, 2, 3, 4, 5, 6, 7, 8, 9, 0], maxlen=10)
>>> dq.appendleft(-1)  ❸
>>> dq
deque([-1, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
>>> dq.extend([11, 22, 33])  ❹
>>> dq
deque([3, 4, 5, 6, 7, 8, 9, 11, 22, 33], maxlen=10)
>>> dq.extendleft([10, 20, 30, 40])  ❺
>>> dq
deque([40, 30, 20, 10, 3, 4, 5, 6, 7, 8], maxlen=10)
```

❶ `maxlen` 是一个可选参数，代表这个队列可以容纳的元素的数量，而

且一旦设定，这个属性就不能修改了。

❷ 队列的旋转操作接受一个参数 n，当 n > 0 时，队列的最右边的 n 个元素会被移动到队列的左边。当 n < 0 时，最左边的 n 个元素会被移动到右边。

❸ 当试图对一个已满（len(d) == d.maxlen）的队列做尾部添加操作的时候，它头部的元素会被删除掉。注意在下一行里，元素 0 被删除了。

❹ 在尾部添加 3 个元素的操作会挤掉 -1、1 和 2。

❺ extendleft(iter) 方法会把迭代器里的元素逐个添加到双向队列的左边，因此迭代器里的元素会逆序出现在队列里。

表 2-3 总结了列表和双向队列这两个类型的方法（object 类包含的方法除外）。

双向队列实现了大部分列表所拥有的方法，也有一些额外的符合自身设计的方法，比如说 popleft 和 rotate。但是为了实现这些方法，双向队列也付出了一些代价，从队列中间删除元素的操作会慢一些，因为它只对在头尾的操作进行了优化。

append 和 popleft 都是原子操作，也就说是 deque 可以在多线程程序中安全地当作先进先出的栈使用，而使用者不需要担心资源锁的问题。

表**2-3**：列表和双向队列的方法（不包括由对象实现的方法）

	列表	双向队列	
s.__add__(s2)	•		s + s2，拼接
s.__iadd__(s2)	•	•	s += s2，就地拼接
			添加一个元素到最右侧（到最后一个元素之

s.append(e)	•	•	后）
s.appendleft(e)		•	添加一个元素到最左侧（到第一个元素之前）
s.clear()	•	•	删除所有元素
s.__contains__(e)	•		s 是否含有 e
s.copy()	•		对列表浅复制
s.__copy__()		•	对 copy.copy（浅复制）的支持
s.count(e)	•	•	s 中 e 出现的次数
s.__delitem__(p)	•	•	把位置 p 的元素移除
s.extend(i)	•	•	将可迭代对象 i 中的元素添加到尾部
s.extendleft(i)		•	将可迭代对象 i 中的元素添加到头部
s.__getitem__(p)	•	•	s[p]，读取位置 p 的元素
s.index(e)	•		找到 e 在序列中第一次出现的位置
s.insert(p, e)	•		在位于 p 的元素之前插入元素 e
s.__iter__()	•	•	返回迭代器
s.__len__()	•	•	len(s)，序列的长度

s.__mul__(n)	•		s * n，重复拼接
s.__imul__(n)	•		s *= n，就地重复拼接
s.__rmul__(n)	•		n * s，反向重复拼接*
s.pop()	•	•	移除最后一个元素并返回它的值#
s.popleft()		•	移除第一个元素并返回它的值
s.remove(e)	•	•	移除序列里第一次出现的 e 元素
s.reverse()	•	•	调转序列中元素的位置
s.__reversed__()	•	•	返回一个从尾部开始扫描元素的迭代器
s.rotate(n)		•	把 n 个元素从队列的一端移到另一端
s.__setitem__(p, e)	•	•	s[p] = e，把位于 p 位置的元素替换成 e
s.sort([key], [revers])	•		就地排序序列，可选参数有 key 和 reverse

* 第 13 章会讲反向运算符。

a_list.pop(p) 这个操作只能用于列表，双向队列的这个方法不接收参数。

除了 deque 之外，还有些其他的 Python 标准库也有对队列的实现。

queue

提供了同步（线程安全）类 Queue、LifoQueue 和 PriorityQueue，不同的线程可以利用这些数据类型来交换信息。这三个类的构造方法都有一个可选参数 maxsize，它接收正整数作为输入值，用来限定队列的大小。但是在满员的时候，这些类不会扔掉旧的元素来腾出位置。相反，如果队列满了，它就会被锁住，直到另外的线程移除了某个元素而腾出了位置。这一特性让这些类很适合用来控制活跃线程的数量。

multiprocessing

这个包实现了自己的 Queue，它跟 queue.Queue 类似，是设计给进程间通信用的。同时还有一个专门的 multiprocessing.JoinableQueue 类型，可以让任务管理变得更方便。

asyncio

Python 3.4 新提供的包，里面有 Queue、LifoQueue、PriorityQueue 和 JoinableQueue，这些类受到 queue 和 multiprocessing 模块的影响，但是为异步编程里的任务管理提供了专门的便利。

heapq

跟上面三个模块不同的是，heapq 没有队列类，而是提供了 heappush 和 heappop 方法，让用户可以把可变序列当作堆队列或者优先队列来使用。

到了这里，我们对列表之外的类的介绍也就告一段落了，是时候阶段性地总结一下对序列类型的探索了。注意我们还没有提到 str（字符串）和二进制序列，它们将在第 4 章中专门介绍。

2.10　本章小结

要想写出准确、高效和地道的 Python 代码，对标准库里的序列类型的掌握是不可或缺的。

Python 序列类型最常见的分类就是可变和不可变序列。但另外一种分类方式也很有用，那就是把它们分为扁平序列和容器序列。前者的体积更小、速度更快而且用起来更简单，但是它只能保存一些原子性的数据，比如数字、字符和字节。容器序列则比较灵活，但是当容器序列遇到可变对象时，用户就需要格外小心了，因为这种组合时常会搞出一些“意外”，特别是带嵌套的数据结构出现时，用户要多费一些心思来保证代码的正确。

列表推导和生成器表达式则提供了灵活构建和初始化序列的方式，这两个工具都异常强大。如果你还不能熟练地使用它们，可以专门花时间练习一下。它们其实不难，而且用起来让人上瘾。

元组在 Python 里扮演了两个角色，它既可以用作无名称的字段的记录，又可以看作不可变的列表。当元组被当作记录来用的时候，拆包是最安全可靠地从元组里提取不同字段信息的方式。新引入的 `*` 句法让元组拆包的便利性更上一层楼，让用户可以选择性忽略不需要的字段。具名元组也已经不是一个新概念了，但它似乎没有受到应有的重视。就像普通元组一样，具名元组的实例也很节省空间，但它同时提供了方便地通过名字来获取元组各个字段信息的方式，另外还有个实用的 `._asdict()` 方法来把记录变成 `OrderedDict` 类型。

Python 里最受欢迎的一个语言特性就是序列切片，而且很多人其实还没完全了解它的强大之处。比如，用户自定义的序列类型也可以选择支持 NumPy 中的多维切片和省略（`...`）。另外，对切片赋值是一个修改可变序列的捷径。

重复拼接 `seq * n` 在正确使用的前提下，能让我们方便地初始化含有不可变元素的多维列表。增量赋值 `+=` 和 `*=` 会区别对待可变和不可变序列。在遇到不可变序列时，这两个操作会在背后生成新的序列。但如果被赋值的对象是可变的，那么这个序列会就地修改——然而这也取决于

序列本身对特殊方法的实现。

序列的 `sort` 方法和内置的 `sorted` 函数虽然很灵活，但是用起来都不难。这两个方法都比较灵活，是因为它们都接受一个函数作为可选参数来指定排序算法如何比较大小，这个参数就是 `key` 参数。`key` 还可以被用在 `min` 和 `max` 函数里。如果在插入新元素的同时还想保持有序序列的顺序，那么需要用到 `bisect.insort`。`bisect.bisect` 的作用则是快速查找。

除了列表和元组，Python 标准库里还有 `array.array`。另外，虽然 NumPy 和 SciPy 都不是 Python 标准库的一部分，但稍微学习一下它们，会让你在处理大规模数值型数据时如有神助。

本章末尾介绍了 `collections.deque` 这个类型，它具有灵活多用和线程安全的特性。表 2-3 将它和列表的 API 做了比较。本章最后也提及了一些标准库中的其他队列类型的实现。

2.11 延伸阅读

David Beazley 和 Brian K. Jones 的《Python Cookbook（第 3 版）中文版》一书的第 1 章"数据结构"里有很多专门针对序列的小窍门。尤其是"1.11 对切片命名"这一部分，从中我学会把切片赋值给变量以改善可读性，本书的示例 2-11 对此做了说明。

《Python Cookbook（第 2 版）中文版》用的是 Python 2.4，但其中大部分的代码都可以运行在 Python 3 中。该书第 5 章和第 6 章的大部分内容都是跟序列有关的。该书的编者有 Alex Martelli、Anna Martelli Ravenscroft 和 David Ascher，另外还有十几位 Python 程序员为内容做出了贡献。第 3 版则是从零开始完全重写的，书的重点也放在了 Python 的语义上，特别是 Python 3 带来的那些变化，而不是像第 2 版那样把重点放在如何解决实际问题上。虽然第 2 版中有些内容已经不是最优解了，但是我仍然推荐把这两个版本都读一读。

Python 官方网站中的"Sorting HOW TO"一文（https://docs.python.org/3/howto/sorting.html）通过几个例子讲解了 `sorted` 和 `list.sort` 的高级用法。

"PEP 3132 — Extended Iterable Unpacking"（https://www.python.org/dev/peps/pep-3132/）算得上是使用 `*extra` 句法进行平行赋值的权威指南。如果你想窥探一下 Python 本身的开发过程，"Missing *-unpacking generalizations"（http://bugs.python.org/issue2292）是一个 bug 追踪器，里面有很多关于如何更广泛地使用可迭代对象拆包的讨论和提议。"PEP 448—Additional Unpacking Generalizations"（https://www.python.org/dev/peps/pep-0448/）就是这些讨论的直接结果。就在我写这本书的时候，这些改动也许会被集成在 Python 3.5 中。

Eli Bendersky 的博客文章"Less Copies in Python with the Buffer Protocol and memoryviews"（http://eli.thegreenplace.net/2011/11/28/less-copies-in-python-with-the-buffer-protocol-and-memoryviews/）里有一些关于 `memoryview` 的小教程。

市面上关于 NumPy 的书多到数不清，其中有些书的名字里都不带“NumPy”这几个字，例如 Wes McKinney 的《利用 Python 进行数据分析》一书。

科学家尤其钟爱 NumPy 和 SciPy 的强大以及与 Python 的交互式控制台的结合，于是他们专门开发了 IPython。IPython 是 Python 自带控制台的强大替代品，而且它还附带了图形界面、内嵌的图表渲染、文学编程支持（代码和文本互动）和 PDF 渲染。而这些互动多媒体对话还能以 IPython 记事本的形式在网络上分享——详见“IPython 记事本”（http://ipython.org/notebook.html）中的截屏和视频。IPython 在 2012 年非常流行，背后的开发者收到了一笔 1 150 000 美元的捐赠。这笔来自 Sloan 基金的捐赠是专门用来支持加州大学伯克利分校的开发者的，好让他们能在 2013—2014 年期间按计划实现 IPython 的扩展。

Python 标准库里的“8.3. collections — Container datatypes”（https://docs.python.org/3/library/collections.html）里有一些关于双向队列和其他集合类型的使用技巧。

Python 里的范围（`range`）和切片都不会返回第二个下标所指的元素，Edsger W. Dijkstra 在一个很短的备忘录里为这一惯例做了最好的辩护。这篇名为“Why Numbering Should Start at Zero”（http://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD8
的备忘录其实是关于数学符号的，但是它跟 Python 的关系在于，Dijkstra 教授严肃又活泼地解释了为什么 2，3，…，12 这个序列应该表达为 $2 \leq i < 13$。备忘录对其他所有的表达习惯都作出了反驳，同时还说明了为什么不能让用户自行决定表达习惯。虽然文章的标题是关于基于 0 的下标，但是整篇文章其实都在说为什么 `'ABCED'[1:3]` 的结果应该是 `'BC'` 而不是 `'BCD'`，以及为什么 2，3，…，12 应该写作 `range(2, 13)`。（顺便说一下，这份备忘录是手写的，但是字写得干净漂亮。如果有人就此创作 Dijkstra 字体，我应该会买一份。）

杂谈

元组的本质

2012 年，我在 PyCon US 上贴了一张关于 ABC 语言的墙报。Guido 在开创 Python 语言之前曾做过 ABC 解释器方面的工作，因此他也去看了我的墙报。我们聊了不少，而且都提到了 ABC 里的

compounds 类型。compounds 算得上是 Python 元组的鼻祖，它既支持平行赋值，又可以用在字典（dict）里作为合成键（ABC 里对应字典的类型是表格，即 table）。但 compounds 不属于序列，它不是迭代类型，也不能通过下标来提取某个值，更不用说切片了。要么把 compounds 对象当作整体来用，要么用平行赋值把里面所有的字段都提取出来，仅此而已。

我跟 Guido 说，上面这些限制让 compounds 的作用变得很明确，它只能用作没有字段名的记录。Guido 回应说，Python 里的元组能当作序列来使用，其实是一个取巧的实现。

这其实体现了 Python 的实用主义，而实用主义是 Python 较之 ABC 更好用也更成功的原因。从一个语言开发人员的角度来看，让元组具有序列的特性可能需要下点功夫，结果则是设计出了一个概念上并不如 compounds 纯粹，却更灵活的元组——它甚至能当成不可变的列表来使用。

说真的，不可变列表这种数据类型在编程语言里真的非常好用（其实 frozenlist 这个名字更酷），而 Python 里这种类型其实就是一个行为很像序列的元组。

“优雅是简约之父”

很久以前，*extra 这种语法就在函数里用来把多个元素赋值给一个参数了。（我有本出版于 1996 年的讲 Python 1.4 的书，里面就提到了这个用法。）Python 1.6 或更新的版本里，这个语法在函数定义里用来把一个可迭代对象拆包成不同的参数，这算是跟上面说的那种用法互补。这一设计直观而优雅，并且取代了 Python 里的 apply 函数。如今到了 Python 3，*extra 这个写法又可以用在赋值表达式的左侧，从而在平行赋值里接收多余的元素。这一点让这个本来就很实用的语法锦上添花。

像这样的改进一个接着一个，让 Python 变得越来越灵活，越来越统一，也越来越简单。“优雅是简约之父”（“Elegance begets simplicity”）是 2009 年在芝加哥的 PyCon 的口号，印在 PyCon 的 T 恤上，同样印在 T 恤上的还有 Bruce Eckel 画的《易经》第二十二卦，即贲卦的卦象。贲代表着典雅高贵。这也是我最喜欢的一件

PyCon 的 T 恤。

扁平序列和容器序列

为了解释不同序列类型里不同的内存模型，我用了容器序列和扁平序列这两个说法。其中“容器”一词来自“Data Model”文档（https://docs.python.org/3/reference/datamodel.html#objects-values-and-types）：

> 有些对象里包含对其他对象的引用；这些对象称为容器。

因此，我特别使用了“容器序列”这个词，因为 Python 里有是容器但并非序列的类型，比如 `dict` 和 `set`。容器序列可以嵌套着使用，因为容器里的引用可以针对包括自身类型在内的任何类型。

与此相反，扁平序列因为只能包含原子数据类型，比如整数、浮点数或字符，所以不能嵌套使用。

称其为“扁平序列”是因为我希望有个名词能够跟“容器序列”形成对比。这个词是我自己发明的，专门用来指代 Python 中“不是容器序列”的序列，在其他地方你可能找不到这样的用法。如果这个词出现在维基百科上面的话，我们需要给它加上“原创研究”标签。我更倾向于把这类词称作“自创名词”，希望它能对你有所帮助并为你所用。

混合类型列表

Python 入门教材往往会强调列表是可以同时容纳不同类型的元素的，但是实际上这样做并没有什么特别的好处。我们之所以用列表来存放东西，是期待在稍后使用它的时候，其中的元素有一些通用的特性（比如，列表里存的是一类可以“呱呱”叫的动物，那么所有的元素都应该会发出这种叫声，即便其中一部分元素类型并不是鸭子）。在 Python 3 中，如果列表里的东西不能比较大小，那么我们就不能对列表进行排序：

```
>>> l = [28, 14, '28', 5, '9', '1', 0, 6, '23', 19]
>>> sorted(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: unorderable types: str() < int()
```

元组则恰恰相反，它经常用来存放不同类型的的元素。这也符合它的本质，元组就是用作存放彼此之间没有关系的数据的记录。

key 参数很妙

`list.sort`、`sorted`、`max` 和 `min` 函数的 `key` 参数是一个很棒的设计。其他语言里的排序函数需要用户提供一个接收两个参数的比较函数作为参数，像是 Python 2 里的 `cmp(a, b)`。用 `key` 参数能把事情变得简单且高效。说它更简单，是因为只需要提供一个单参数函数来提取或者计算一个值作为比较大小的标准即可，而 Python 2 的这种设计则需要用户写一个返回值是—1、0 或者 1 的双参数函数。说它更高效，是因为在每个元素上，`key` 函数只会被调用一次。而双参数比较函数则在每一次两两比较的时候都会被调用。诚然，在排序的时候，Python 总会比较两个键（`key`），但是那一阶段的计算会发生在 C 语言那一层，这样会比调用用户自定义的 Python 比较函数更快。

另外，`key` 参数也能让你对一个混有数字字符和数值的列表进行排序。你只需要决定到底是把字符看作数值，还是把数值看作字符：

```
>>> l = [28, 14, '28', 5, '9', '1', 0, 6, '23', 19]
>>> sorted(l, key=int)
[0, '1', 5, 6, '9', 14, 19, '23', 28, '28']
>>> sorted(l, key=str)
[0, '1', 14, 19, '23', 28, '28', 5, 6, '9']
```

Oracle、Google 和 Timbot 之间的八卦

`sorted` 和 `list.sort` 背后的排序算法是 Timsort，它是一种自适应算法，会根据原始数据的顺序特点交替使用插入排序和归并排序，以达到最佳效率。这样的算法被证明是很有效的，因为来自真实世界的数据通常是有一定的顺序特点的。维基百科上有一个条目是关于这个算法的（https://en.wikipedia.org/wiki/Timsort）。

Timsort 在 2002 年的时候首次用在 CPython 中；自 2009 年起，Java

和 Android 也开始使用这个算法。后面这个时间点如此广为人知，是因为在 Google 对 Sun 的侵权案中， Oracle 把 Timsort 中的一些相关代码当作了呈堂证供。详见“ Oracle v. Google—Day 14 Filings”一文（http://www.groklaw.net/articlebasic.php?story=20120510205659643）。

Timsort 的创始人是 Tim Peters，他同时也是一位高产的 Python 核心开发者。由于他贡献了太多代码，以至于很多人都说他其实是人工智能，他也就有了“Timbot”这一绰号。在“Python Humor”（https://www.python.org/doc/humor/#id9）里可以读到相关的故事。Tim 也是“Python 之禅”（`import this`）的作者。

第 3 章　字典和集合

> 字典这个数据结构活跃在所有 Python 程序的背后，即便你的源码里并没有直接用到它。
>
> ——A. M. Kuchling
> 《代码之美》第 18 章“Python 的字典类：如何打造全能战士”

`dict` 类型不但在各种程序里广泛使用，它也是 Python 语言的基石。模块的命名空间、实例的属性和函数的关键字参数中都可以看到字典的身影。跟它有关的内置函数都在 `__builtins__.__dict__` 模块中。

正是因为字典至关重要，Python 对它的实现做了高度优化，而散列表则是字典类型性能出众的根本原因。

集合（`set`）的实现其实也依赖于散列表，因此本章也会讲到它。反过来说，想要进一步理解集合和字典，就得先理解散列表的原理。

本章内容的大纲如下：

- 常见的字典方法
- 如何处理查找不到的键
- 标准库中 `dict` 类型的变种
- `set` 和 `frozenset` 类型
- 散列表的工作原理
- 散列表带来的潜在影响（什么样的数据类型可作为键、不可预知的顺序，等等）

3.1 泛映射类型

`collections.abc` 模块中有 `Mapping` 和 `MutableMapping` 这两个抽象基类，它们的作用是为 `dict` 和其他类似的类型定义形式接口（在 Python 2.6 到 Python 3.2 的版本中，这些类还不属于 `collections.abc` 模块，而是隶属于 `collections` 模块）。详见图 3-1。

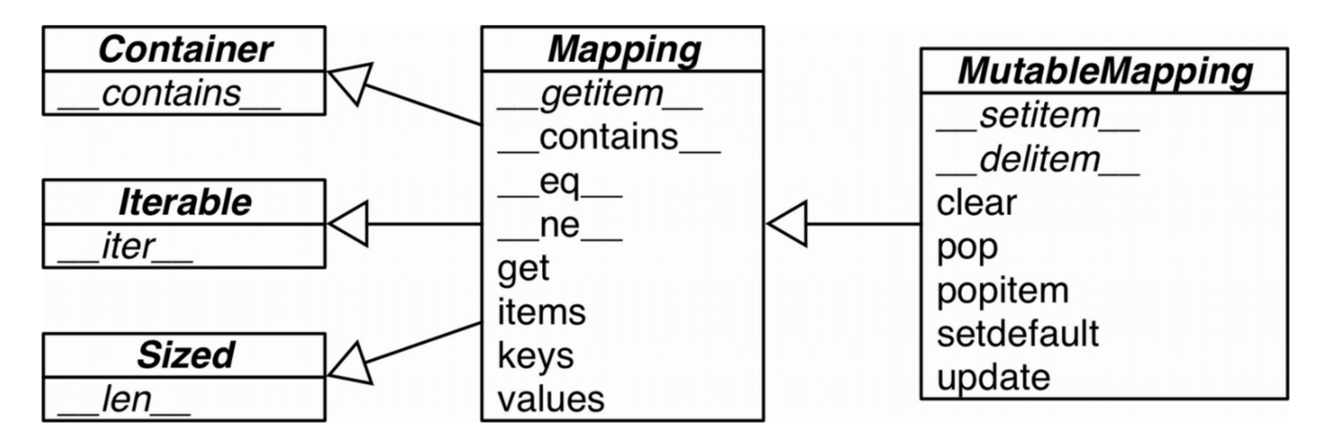


图 **3-1**：**`collections.abc`** 中的 **`MutableMapping`** 和它的超类的 **UML** 类图（箭头从子类指向超类，抽象类和抽象方法的名称以斜体显示）

然而，非抽象映射类型一般不会直接继承这些抽象基类，它们会直接对 `dict` 或是 `collections.User.Dict` 进行扩展。这些抽象基类的主要作用是作为形式化的文档，它们定义了构建一个映射类型所需要的最基本的接口。然后它们还可以跟 `isinstance` 一起被用来判定某个数据是不是广义上的映射类型：

```
>>> my_dict = {}
>>> isinstance(my_dict, abc.Mapping)
True
```

这里用 `isinstance` 而不是 `type` 来检查某个参数是否为 `dict` 类型，因为这个参数有可能不是 `dict`，而是一个比较另类的映射类型。

标准库里的所有映射类型都是利用 `dict` 来实现的，因此它们有个共同

的限制，即只有可散列的数据类型才能用作这些映射里的键（只有键有这个要求，值并不需要是可散列的数据类型）。

什么是可散列的数据类型

在 Python 词汇表（https://docs.python.org/3/glossary.html#term-hashable）中，关于可散列类型的定义有这样一段话：

> 如果一个对象是可散列的，那么在这个对象的生命周期中，它的散列值是不变的，而且这个对象需要实现 `__hash__()` 方法。另外可散列对象还要有 `__qe__()` 方法，这样才能跟其他键做比较。如果两个可散列对象是相等的，那么它们的散列值一定是一样的……

原子不可变数据类型（`str`、`bytes` 和数值类型）都是可散列类型，`frozenset` 也是可散列的，因为根据其定义，`frozenset` 里只能容纳可散列类型。元组的话，只有当一个元组包含的所有元素都是可散列类型的情况下，它才是可散列的。来看下面的元组 `tt`、`tl` 和 `tf`：

```
>>> tt = (1, 2, (30, 40))
>>> hash(tt)
8027212646858338501
>>> tl = (1, 2, [30, 40])
>>> hash(tl)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> tf = (1, 2, frozenset([30, 40]))
>>> hash(tf)
-4118419923444501110
```

直到我写这本书的时候，Python 词汇表（https://docs.python.org/3/glossary.html#term-hashable）里还在说“Python 里所有的不可变类型都是可散列的”。这个说法其实是不准确的，比如虽然元组本身是不可变序列，它里面的元素可能是其他可变类型的引用。

一般来讲用户自定义的类型的对象都是可散列的，散列值就是它们的 `id()` 函数的返回值，所以所有这些对象在比较的时候都是不相等的。如果一个对象实现了 `__eq__` 方法，并且在方法中用到了这个对象的内部状态的话，那么只有当所有这些内部状态都是不可变的情况下，这个对象才是可散列的。

根据这些定义，字典提供了很多种构造方法，“Built-in Types”（https://docs.python.org/3/library/stdtypes.html#mapping-types-dict）这个页面上有个例子来说明创建字典的不同方式：

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

除了这些字面句法和灵活的构造方法之外，字典推导（dict comprehension）也可以用来建造新 `dict`，详见下一节。

3.2 字典推导

自 Python 2.7 以来，列表推导和生成器表达式的概念就移植到了字典上，从而有了字典推导（后面还会看到集合推导）。字典推导（dictcomp）可以从任何以键值对作为元素的可迭代对象中构建出字典。示例 3-1 就展示了利用字典推导可以把一个装满元组的列表变成两个不同的字典。

示例 **3-1** 字典推导的应用

```
>>> DIAL_CODES = [                  ❶
...         (86, 'China'),
...         (91, 'India'),
...         (1, 'United States'),
...         (62, 'Indonesia'),
...         (55, 'Brazil'),
...         (92, 'Pakistan'),
...         (880, 'Bangladesh'),
...         (234, 'Nigeria'),
...         (7, 'Russia'),
...         (81, 'Japan'),
...     ]
>>> country_code = {country: code for code, country in DIAL_CODES}  ❷
>>> country_code
{'China': 86, 'India': 91, 'Bangladesh': 880, 'United States': 1,
'Pakistan': 92, 'Japan': 81, 'Russia': 7, 'Brazil': 55, 'Nigeria':
234, 'Indonesia': 62}
>>> {code: country.upper() for country, code in country_code.items()  ❸
...   if code < 66}
{1: 'UNITED STATES', 55: 'BRAZIL', 62: 'INDONESIA', 7: 'RUSSIA'}
```

❶ 一个承载成对数据的列表，它可以直接用在字典的构造方法中。

❷ 这里把配好对的数据左右换了下，国家名是键，区域码是值。

❸ 跟上面相反，用区域码作为键，国家名称转换为大写，并且过滤掉区域码大于或等于 66 的地区。

如果列表推导的概念已经为你所熟知，接受字典推导应该不难。如果你对列表推导还不熟，那么是时候来掌握它了，因为字典推导的表达形式

会蔓延到其他数据类型中。

下面来看看映射类型提供的 API 的全景图。

3.3 常见的映射方法

映射类型的方法其实很丰富。表 3-1 为我们展示了 dict、defaultdict 和 OrderedDict 的常见方法，后面两个数据类型是 dict 的变种，位于 collections 模块内。

表3-1：**dict**、**collections.defaultdict**和**collections.OrderedDict**这三种映射类型的方法列表（依然省略了继承自**object**的常见方法）；可选参数以**[...]**表示

	dict	defaultdict	OrderedDict	
d.clear()	•	•	•	移除所有元素
d.__contains__(k)	•	•	•	检查 k 是否在 d 中
d.copy()	•	•	•	浅复制
d.__copy__()		•		用于支持 copy.copy
d.default_factory		•		在 __missing__ 函数中被调用的函数，用以给未找到的元素设置值*
d.__delitem__(k)	•	•	•	del d[k]，移除键为 k 的元素
d.fromkeys(it, [initial])	•	•	•	将迭代器 it 里的元素设置为映射里的键，如果有 initial 参数，就把它作为这些键对应的值（默认是 None）
				返回键 k 对应的值，如果字典里

d.get(k, [default])	•	•	•	没有键 k，则返回 None 或者 default
d.__getitem__(k)	•	•	•	让字典 d 能用 d[k] 的形式返回键 k 对应的值
d.items()	•	•	•	返回 d 里所有的键值对
d.__iter__()	•	•	•	获取键的迭代器
d.keys()	•	•	•	获取所有的键
d.__len__()	•	•	•	可以用 len(d) 的形式得到字典里键值对的数量
d.__missing__(k)		•		当 __getitem__ 找不到对应键的时候，这个方法会被调用
d.move_to_end(k, [last])			•	把键为 k 的元素移动到最靠前或者最靠后的位置（last 的默认值是 True）
d.pop(k, [defaul]	•	•	•	返回键 k 所对应的值，然后移除这个键值对。如果没有这个键，返回 None 或者 defaul
d.popitem()	•	•	•	随机返回一个键值对并从字典里移除它[#]
d.__reversed__()			•	返回倒序的键的迭代器
d.setdefault(k, [default])	•	•	•	若字典里有键k，则把它对应的值设置为 default，然后返回这个值；若无，则让 d[k] =

				default，然后返回 default
d.__setitem__(k, v)	•	•	•	实现 d[k] = v 操作，把 k 对应的值设为v
d.update(m, [**kargs])	•	•	•	m 可以是映射或者键值对迭代器，用来更新 d 里对应的条目
d.values()	•	•	•	返回字典里的所有值

* default_factory 并不是一个方法，而是一个可调用对象（callable），它的值在 defaultdict 初始化的时候由用户设定。

OrderedDict.popitem() 会移除字典里最先插入的元素（先进先出）；同时这个方法还有一个可选的 last 参数，若为真，则会移除最后插入的元素（后进先出）。

上面的表格中，update 方法处理参数 m 的方式，是典型的“鸭子类型”。函数首先检查 m 是否有 keys 方法，如果有，那么 update 函数就把它当作映射对象来处理。否则，函数会退一步，转而把 m 当作包含了键值对 (key, value) 元素的迭代器。Python 里大多数映射类型的构造方法都采用了类似的逻辑，因此你既可以用一个映射对象来新建一个映射对象，也可以用包含 (key, value) 元素的可迭代对象来初始化一个映射对象。

在映射对象的方法里，setdefault 可能是比较微妙的一个。我们虽然并不会每次都用它，但是一旦它发挥作用，就可以节省不少次键查询，从而让程序更高效。如果你对它还不熟悉，下面我会通过一个实例来讲解它的用法。

用**setdefault**处理找不到的键

当字典 d[k] 不能找到正确的键的时候，Python 会抛出异常，这个行为符合 Python 所信奉的“快速失败”哲学。也许每个 Python 程序员都知道可以用 d.get(k, default) 来代替 d[k]，给找不到的键一个默认的返回值（这比处理 KeyError 要方便不少）。但是要更新某个键对应的

值的时候，不管使用 `__getitem__` 还是 `get` 都会不自然，而且效率低。就像示例 3-2 中的还没有经过优化的代码所显示的那样，`dict.get` 并不是处理找不到的键的最好方法。

示例 3-2 是由 Alex Martelli 举的一个例子 [1] 变化而来，例子生成的索引跟示例 3-3 显示的一样。

> [1]示例代码出现在 Martelli 的演讲“Re-learning python”中（第 41 张幻灯片，http://www.aleax.it/Python/accu04_Relearn_Python_alex.pdf），他的代码被我放在了示例 3-4 中，代码很好地展示了 `dict.setdefault` 的用法。

示例 **3-2** `index0.py` 这段程序从索引中获取单词出现的频率信息，并把它们写进对应的列表里（更好的解决方案在示例 3-4 中）

```
"""创建一个从单词到其出现情况的映射"""

import sys
import re

WORD_RE = re.compile(r'\w+')

index = {}
with open(sys.argv[1], encoding='utf-8') as fp:
    for line_no, line in enumerate(fp, 1):
        for match in WORD_RE.finditer(line):
            word = match.group()
            column_no = match.start()+1
            location = (line_no, column_no)
            # 这其实是一种很不好的实现，这样写只是为了证明论点
            occurrences = index.get(word, [])  ❶
            occurrences.append(location)       ❷
            index[word] = occurrences          ❸
            # 以字母顺序打印出结果
for word in sorted(index, key=str.upper):      ❹
    print(word, index[word])
```

❶ 提取 `word` 出现的情况，如果还没有它的记录，返回 `[]`。

❷ 把单词新出现的位置添加到列表的后面。

❸ 把新的列表放回字典中，这又牵扯到一次查询操作。

❹ sorted 函数的 key= 参数没有调用 str.uppper，而是把这个方法的引用传递给 sorted 函数，这样在排序的时候，单词会被规范成统一格式。[2]

> [2]这是将方法用作一等函数的一个示例，第 5 章会谈到这一点。

示例 **3-3**　这里是示例3-2 的不完全输出，每一行的列表都代表一个单词的出现情况，列表中的元素是一对值，第一个值表示出现的行，第二个表示出现的列

```
$ python3 index0.py ../../data/zen.txt
a [(19, 48), (20, 53)]
Although [(11, 1), (16, 1), (18, 1)]
ambiguity [(14, 16)]
and [(15, 23)]
are [(21, 12)]
aren [(10, 15)]
at [(16, 38)]
bad [(19, 50)]
be [(15, 14), (16, 27), (20, 50)]
beats [(11, 23)]
Beautiful [(3, 1)]
better [(3, 14), (4, 13), (5, 11), (6, 12), (7, 9), (8, 11),
(17, 8), (18, 25)]
...
```

示例 3-2 里处理单词出现情况的三行，通过 dict.setdefault 可以只用一行解决。示例 3-4 更接近 Alex Martelli 自己举的例子。

示例 **3-4**　index.py 用一行就解决了获取和更新单词的出现情况列表，当然跟示例 3-2 不一样的是，这里用到了 dict.setdefault

```
"""创建从一个单词到其出现情况的映射"""

import sys
import re

WORD_RE = re.compile(r'\w+')

index = {}
with open(sys.argv[1], encoding='utf-8') as fp:
```

```
    for line_no, line in enumerate(fp, 1):
        for match in WORD_RE.finditer(line):
            word = match.group()
            column_no = match.start()+1
            location = (line_no, column_no)
            index.setdefault(word, []).append(location)  ❶

# 以字母顺序打印出结果
for word in sorted(index, key=str.upper):
    print(word, index[word])
```

❶ 获取单词的出现情况列表，如果单词不存在，把单词和一个空列表放进映射，然后返回这个空列表，这样就能在不进行第二次查找的情况下更新列表了。

也就是说，这样写：

```
my_dict.setdefault(key, []).append(new_value)
```

跟这样写：

```
if key not in my_dict:
    my_dict[key] = []
my_dict[key].append(new_value)
```

二者的效果是一样的，只不过后者至少要进行两次键查询——如果键不存在的话，就是三次，用 `setdefault` 只需要一次就可以完成整个操作。

那么，在单纯地查找取值（而不是通过查找来插入新值）的时候，该怎么处理找不到的键呢？

3.4 映射的弹性键查询

有时候为了方便起见，就算某个键在映射里不存在，我们也希望在通过这个键读取值的时候能得到一个默认值。有两个途径能帮我们达到这个目的，一个是通过 `defaultdict` 这个类型而不是普通的 `dict`，另一个是给自己定义一个 `dict` 的子类，然后在子类中实现 `__missing__` 方法。下面将介绍这两种方法。

3.4.1 `defaultdict`：处理找不到的键的一个选择

示例 3-5 在 `collections.defaultdict` 的帮助下优雅地解决了示例 3-4 里的问题。在用户创建 `defaultdict` 对象的时候，就需要给它配置一个为找不到的键创造默认值的方法。

具体而言，在实例化一个 `defaultdict` 的时候，需要给构造方法提供一个可调用对象，这个可调用对象会在 `__getitem__` 碰到找不到的键的时候被调用，让 `__getitem__` 返回某种默认值。

比如，我们新建了这样一个字典：`dd = defaultdict(list)`，如果键 `'new-key'` 在 `dd` 中还不存在的话，表达式 `dd['new-key']` 会按照以下的步骤来行事。

(1) 调用 `list()` 来建立一个新列表。

(2) 把这个新列表作为值，`'new-key'` 作为它的键，放到 `dd` 中。

(3) 返回这个列表的引用。

而这个用来生成默认值的可调用对象存放在名为 `default_factory` 的实例属性里。

示例 **3-5**　index_default.py：利用 `defaultdict` 实例而不是 `setdefault` 方法

```
"""创建一个从单词到其出现情况的映射"""
```

```
import sys
import re
import collections

WORD_RE = re.compile(r'\w+')

index = collections.defaultdict(list)     ❶
with open(sys.argv[1], encoding='utf-8') as fp:
    for line_no, line in enumerate(fp, 1):
        for match in WORD_RE.finditer(line):
            word = match.group()
            column_no = match.start()+1
            location = (line_no, column_no)
            index[word].append(location)   ❷

# 以字母顺序打印出结果
for word in sorted(index, key=str.upper):
    print(word, index[word])
```

❶ 把 list 构造方法作为 default_factory 来创建一个 defaultdict。

❷ 如果 index 并没有 word 的记录，那么 default_factory 会被调用，为查询不到的键创造一个值。这个值在这里是一个空的列表，然后这个空列表被赋值给 index[word]，继而被当作返回值返回，因此 .append(location) 操作总能成功。

如果在创建 defaultdict 的时候没有指定 default_factory，查询不存在的键会触发 KeyError。

> defaultdict 里的 default_factory 只会在 __getitem__ 里被调用，在其他的方法里完全不会发挥作用。比如，dd 是个 defaultdict，k 是个找不到的键， dd[k] 这个表达式会调用 default_factory 创造某个默认值，而 dd.get(k) 则会返回 None。

所有这一切背后的功臣其实是特殊方法 __missing__。它会在 defaultdict 遇到找不到的键的时候调用 default_factory，而实际

上这个特性是所有映射类型都可以选择去支持的。

3.4.2　特殊方法__missing__

所有的映射类型在处理找不到的键的时候，都会牵扯到 `__missing__` 方法。这也是这个方法称作“missing”的原因。虽然基类 `dict` 并没有定义这个方法，但是 `dict` 是知道有这么个东西存在的。也就是说，如果有一个类继承了 `dict`，然后这个继承类提供了 `__missing__` 方法，那么在 `__getitem__` 碰到找不到的键的时候，Python 就会自动调用它，而不是抛出一个 `KeyError` 异常。

> `__missing__` 方法只会被 `__getitem__` 调用（比如在表达式 `d[k]` 中）。提供 `__missing__` 方法对 `get` 或者 `__contains__`（`in` 运算符会用到这个方法）这些方法的使用没有影响。这也是我在上一节最后的警告中提到，`defaultdict` 中的 `default_factory` 只对 `__getitem__` 有作用的原因。

有时候，你会希望在查询的时候，映射类型里的键统统转换成 `str`。为可编程电路板（像 Raspberry Pi 或 Arduino[3]）准备的 Pingo.io（http://www.pingo.io/docs/）项目里就有具体的例子。在 Pingo.io 里，电路板上的 GPIO 针脚[4]以 `board.pins` 为名，封装在名为 `board` 的对象里。`board.pins` 是一个映射类型，其中键是针脚的物理位置，它可能只是一个数字或字符串，比如 `"A0"` 或 `"P9_12"`；值则是针脚连接的东西。为了保持一致性，我们希望 `board.pins` 的键只能是字符串，但是为了方便查询，`my_arduino.pins[13]` 也是可行的，这样可以帮 Arduino 的初级玩家快速找到第 13 个针脚上的 LED 灯。示例 3-6 展示了这样的一个映射是怎么运行的。

> [3]Raspberry Pi 是一个集成到巴掌大小的板子上的电脑。Arduino 则是一种可以在烧录程序的同时，连接上各种传感器，用以跟物理世界交互的电路板。更多的相关信息可以在 https://www.raspberrypi.org/ 和 https://www.arduino.cc/ 上找到。——译者注

> [4]通用输入输出针脚，用来跟传感器或其他设备用数据互动。——译者注

示例 **3-6**　当有非字符串的键被查找的时候，`StrKeyDict0` 是如何在该键不存在的情况下，把它转换为字符串的

```
Tests for item retrieval using `d[key]` notation::

    >>> d = StrKeyDict0([('2', 'two'), ('4', 'four')])
    >>> d['2']
    'two'
    >>> d[4]
    'four'
    >>> d[1]
    Traceback (most recent call last):
      ...
    KeyError: '1'

Tests for item retrieval using `d.get(key)` notation::

    >>> d.get('2')
    'two'
    >>> d.get(4)
    'four'
    >>> d.get(1, 'N/A')
    'N/A'

Tests for the `in` operator::

    >>> 2 in d
    True
    >>> 1 in d
    False
```

示例 3-7 则实现了上面例子里的 `StrKeyDict0` 类。

如果要自定义一个映射类型，更合适的策略其实是继承 `collections.UserDict` 类（示例 3-8 就是如此）。这里我们从 `dict` 继承，只是为了演示 `__missing__` 是如何被 `dict.__getitem__` 调用的。

示例 **3-7** `StrKeyDict0` 在查询的时候把非字符串的键转换为字符串

```
class StrKeyDict0(dict):  ❶

    def __missing__(self, key):
        if isinstance(key, str):  ❷
```

```
            raise KeyError(key)
        return self[str(key)]  ❸

    def get(self, key, default=None):
        try:
            return self[key]  ❹
    except KeyError:
        return default  ❺

def __contains__(self, key):
    return key in self.keys() or str(key) in self.keys()  ❻
```

❶ StrKeyDict0 继承了 dict。

❷ 如果找不到的键本身就是字符串，那就抛出 KeyError 异常。

❸ 如果找不到的键不是字符串，那么把它转换成字符串再进行查找。

❹ get 方法把查找工作用 self[key] 的形式委托给 __getitem__，这样在宣布查找失败之前，还能通过 __missing__ 再给某个键一个机会。

❺ 如果抛出 KeyError，那么说明 __missing__ 也失败了，于是返回 default。

❻ 先按照传入键的原本的值来查找（我们的映射类型中可能含有非字符串的键），如果没找到，再用 str() 方法把键转换成字符串再查找一次。

下面来看看为什么 isinstance(key, str) 测试在上面的 __missing__ 中是必需的。

如果没有这个测试，只要 str(k) 返回的是一个存在的键，那么 __missing__ 方法是没问题的，不管是字符串键还是非字符串键，它都能正常运行。但是如果 str(k) 不是一个存在的键，代码就会陷入无限递归。这是因为 __missing__ 的最后一行中的 self[str(key)] 会调用 __getitem__，而这个 str(key) 又不存在，于是 __missing__ 又会被调用。

为了保持一致性，`__contains__` 方法在这里也是必需的。这是因为 `k in d` 这个操作会调用它，但是我们从 `dict` 继承到的 `__contains__` 方法不会在找不到键的时候调用 `__missing__` 方法。`__contains__` 里还有个细节，就是我们这里没有用更具 Python 风格的方式——`k in my_dict`——来检查键是否存在，因为那也会导致 `__contains__` 被递归调用。为了避免这一情况，这里采取了更显式的方法，直接在这个 `self.keys()` 里查询。

> 像 `k in my_dict.keys()` 这种操作在 Python 3 中是很快的，而且即便映射类型对象很庞大也没关系。这是因为 `dict.keys()` 的返回值是一个“视图”。视图就像一个集合，而且跟字典类似的是，在视图里查找一个元素的速度很快。在“Dictionary view objects”（https://docs.python.org/3/library/stdtypes.html#dictionary-view-objects）里可以找到关于这个细节的文档。Python 2 的 `dict.keys()` 返回的是个列表，因此虽然上面的方法仍然是正确的，它在处理体积大的对象的时候效率不会太高，因为 `k in my_list` 操作需要扫描整个列表。

出于对准确度的考虑，我们也需要这个按照键的原本的值来查找的操作（也就是 `key in self.keys()`），因为在创建 `StrKeyDict0` 和为它添加新值的时候，我们并没有强制要求传入的键必须是字符串。因为这个操作没有规定死键的类型，所以让查找操作变得更加友好。

好了，我们已经见识过 `dict` 和 `defaultdict` 了。但是标准库里面还有很多其他的映射类型，下面就来看看。

3.5 字典的变种

这一节总结了标准库里 collections 模块中，除了 defaultdict 之外的不同映射类型。

collections.OrderedDict

这个类型在添加键的时候会保持顺序，因此键的迭代次序总是一致的。OrderedDict 的 popitem 方法默认删除并返回的是字典里的最后一个元素，但是如果像 my_odict.popitem(last=False) 这样调用它，那么它删除并返回第一个被添加进去的元素。

collections.ChainMap

该类型可以容纳数个不同的映射对象，然后在进行键查找操作的时候，这些对象会被当作一个整体被逐个查找，直到键被找到为止。这个功能在给有嵌套作用域的语言做解释器的时候很有用，可以用一个映射对象来代表一个作用域的上下文。在 collections 文档介绍 ChainMap 对象的那一部分（https://docs.python.org/3/library/collections.html#collections.ChainMap）里有一些具体的使用示例，其中包含了下面这个 Python 变量查询规则的代码片段：

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

collections.Counter

这个映射类型会给键准备一个整数计数器。每次更新一个键的时候都会增加这个计数器。所以这个类型可以用来给可散列表对象计数，或者是当成多重集来用——多重集合就是集合里的元素可以出现不止一次。Counter 实现了 + 和 - 运算符用来合并记录，还有像 most_common([n]) 这类很有用的方法。most_common([n]) 会按照次序返回映射里最常见的 n 个键和它们的计数，详情参阅文档（https://docs.python.org/3/library/collections.html#collections.Counter）。

下面的小例子利用 Counter 来计算单词中各个字母出现的次数：

```
>>> ct = collections.Counter('abracadabra')
>>> ct
Counter({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
>>> ct.update('aaaaazzz')
>>> ct
Counter({'a': 10, 'z': 3, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
>>> ct.most_common(2)
[('a', 10), ('z', 3)]
```

colllections.UserDict

这个类其实就是把标准 dict 用纯 Python 又实现了一遍。

跟 OrderedDict、ChainMap 和 Counter 这些开箱即用的类型不同，UserDict 是让用户继承写子类的。下面就来试试。

3.6 子类化UserDict

就创造自定义映射类型来说，以 UserDict 为基类，总比以普通的 dict 为基类要来得方便。

这体现在，我们能够改进示例 3-7 中定义的 StrKeyDict0 类，使得所有的键都存储为字符串类型。

而更倾向于从 UserDict 而不是从 dict 继承的主要原因是，后者有时会在某些方法的实现上走一些捷径，导致我们不得不在它的子类中重写这些方法，但是 UserDict 就不会带来这些问题。[5]

> [5]关于从 dict 或者其他内置类继承到底有什么不好，详见 12.1 节。

另外一个值得注意的地方是，UserDict 并不是 dict 的子类，但是 UserDict 有一个叫作 data 的属性，是 dict 的实例，这个属性实际上是 UserDict 最终存储数据的地方。这样做的好处是，比起示例 3-7，UserDict 的子类就能在实现 __setitem__ 的时候避免不必要的递归，也可以让 __contains__ 里的代码更简洁。

多亏了 UserDict，示例 3-8 里的 StrKeyDict 的代码比示例 3-7 里的 StrKeyDict0 要短一些，功能却更完善：它不但把所有的键都以字符串的形式存储，还能处理一些创建或者更新实例时包含非字符串类型的键这类意外情况。

示例 **3-8**　无论是添加、更新还是查询操作，StrKeyDict 都会把非字符串的键转换为字符串

```
import collections

class StrKeyDict(collections.UserDict):  ❶

    def __missing__(self, key):  ❷
        if isinstance(key, str):
            raise KeyError(key)
        return self[str(key)]
```

```
    def __contains__(self, key):
        return str(key) in self.data  ❸

    def __setitem__(self, key, item):
        self.data[str(key)] = item  ❹
```

❶ StrKeyDict 是对 UserDict 的扩展。

❷ __missing__ 跟示例 3-7 里的一模一样。

❸ __contains__ 则更简洁些。这里可以放心假设所有已经存储的键都是字符串。因此，只要在 self.data 上查询就好了，并不需要像 StrKeyDict0 那样去麻烦 self.keys()。

❹ __setitem__ 会把所有的键都转换成字符串。由于把具体的实现委托给了 self.data 属性，这个方法写起来也不难。

因为 UserDict 继承的是 MutableMapping，所以 StrKeyDict 里剩下的那些映射类型的方法都是从 UserDict、MutableMapping 和 Mapping 这些超类继承而来的。特别是最后的 Mapping 类，它虽然是一个抽象基类（ABC），但它却提供了好几个实用的方法。以下两个方法值得关注。

MutableMapping.update

　　这个方法不但可以为我们所直接利用，它还用在 __init__ 里，让构造方法可以利用传入的各种参数（其他映射类型、元素是 (key, value) 对的可迭代对象和键值参数）来新建实例。因为这个方法在背后是用 self[key] = value 来添加新值的，所以它其实是在使用我们的 __setitem__ 方法。

Mapping.get

　　在 StrKeyDict0（示例 3-7）中，我们不得不改写 get 方法，好让它的表现跟 __getitem__ 一致。而在示例 3-8 中就没这个必要了，因为它继承了 Mapping.get 方法，而 Python 的源码（https://hg.python.org/cpython/file/3.4/Lib/_collections_abc.py#l422）显

示，这个方法的实现方式跟 `StrKeyDict0.get` 是一模一样的。

> 在写完 `StrKeyDict` 这个类之后，我读到了 Antonie Pitrou 写的“PEP 455 — Adding a key-transforming dictionary to collections”（https://www.python.org/dev/peps/pep-0455/）。文章附带的补丁里包含了一个叫作 `TransformDict` 的新类型。这个补丁通过 issue 18986（http://bugs.python.org/issue18986）被吸收进了 Python 3.5。为了试试这个类，我把它提取出来放进了一个单独的模块（在本书代码仓库中：03-dict-set/transformdict.py，https://github.com/fluentpython/example-code/blob/master/03-dict-set/transformdict.py）。比起 `StrKeyDict`，`TransformDict` 的通用性更强，也更复杂，因为它把键存成字符串的同时，还要按照它原来的样子存一份。

之前我们见识过了不可变的序列类型，那有没有不可变的字典类型呢？这么说吧，在标准库里是没有这样的类型的，但是可以用替身来代替。

3.7　不可变映射类型

标准库里所有的映射类型都是可变的，但有时候你会有这样的需求，比如不能让用户错误地修改某个映射。3.4.2 节提到过 Pingo.io，它里面就有个现成的例子。Pingo.io 里有个映射的名字叫作 `board.pins`，里面的数据是 GPIO 物理针脚的信息，我们当然不希望用户一个疏忽就把这些信息给改了。因为硬件方面的东西是不会受软件影响的，所以如果把这个映射里的信息改了，就跟物理上的元件对不上号了。

从 Python 3.3 开始，`types` 模块中引入了一个封装类名叫 `MappingProxyType`。如果给这个类一个映射，它会返回一个只读的映射视图。虽然是个只读视图，但是它是动态的。这意味着如果对原映射做出了改动，我们通过这个视图可以观察到，但是无法通过这个视图对原映射做出修改。示例 3-9 简短地对这个类的用法做了个演示。

示例 **3-9**　用 `MappingProxyType` 来获取字典的只读实例 `mappingproxy`

```
>>> from types import MappingProxyType
>>> d = {1:'A'}
>>> d_proxy = MappingProxyType(d)
>>> d_proxy
mappingproxy({1: 'A'})
>>> d_proxy[1]  ❶
'A'
>>> d_proxy[2] = 'x'  ❷
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'mappingproxy' object does not support item assignment
>>> d[2] = 'B'
>>> d_proxy  ❸
mappingproxy({1: 'A', 2: 'B'})
>>> d_proxy[2]
'B'
>>>
```

❶ `d` 中的内容可以通过 `d_proxy` 看到。

❷ 但是通过 `d_proxy` 并不能做任何修改。

❸ `d_proxy` 是动态的，也就是说对 `d` 所做的任何改动都会反馈到它上面。

因此在 Pingo.io 中我们是这样用它的：`Board` 的具体子类会提供一个包含针脚信息的私有映射成员，然后通过公开属性 `.pins` 把这个映射暴露给 API 的客户，而 `.pins` 属性其实就是用 `mappingproxy` 实现的。一旦这样写好了，客户就不能对这个映射进行任何意外的添加、移除或者修改操作。[6]

> [6]为了照顾 Python 2.7，现实中的 Pingo.io 没有借用 `MappingProxyType` 来实现这个功能，因为它只在 Python 3.3 里才有。

到了这里，我们对标准库中的大多数映射类型都有了一些了解，下面让我们移步到集合类型。

3.8　集合论

“集”这个概念在 Python 中算是比较年轻的，同时它的使用率也比较低。`set` 和它的不可变的姊妹类型 `frozenset` 直到 Python 2.3 才首次以模块的形式出现，然后在 Python 2.6 中它们升级成为内置类型。

> 本书中“集”或者“集合”既指 `set`，也指 `frozenset`。当“集”仅指代 `set` 类时，我会用等宽字体表示[7]。

[7]“集”在英文中就是 set，因此原书中需要用等宽字体来区分特指和泛指。——编者注

集合的本质是许多唯一对象的聚集。因此，集合可以用于去重：

```
>>> l = ['spam', 'spam', 'eggs', 'spam']
>>> set(l)
{'eggs', 'spam'}
>>> list(set(l))
['eggs', 'spam']
```

集合中的元素必须是可散列的，`set` 类型本身是不可散列的，但是 `frozenset` 可以。因此可以创建一个包含不同 `frozenset` 的 `set`。

除了保证唯一性，集合还实现了很多基础的中缀运算符。给定两个集合 `a` 和 `b`，`a | b` 返回的是它们的合集，`a & b` 得到的是交集，而 `a - b` 得到的是差集。合理地利用这些操作，不仅能够让代码的行数变少，还能减少 Python 程序的运行时间。这样做同时也是为了让代码更易读，从而更容易判断程序的正确性，因为利用这些运算符可以省去不必要的循环和逻辑操作。

例如，我们有一个电子邮件地址的集合（`haystack`），还要维护一个较小的电子邮件地址集合（`needles`），然后求出 `needles` 中有多少地址同时也出现在了 `heystack` 里。借助集合操作，我们只需要一行代码就可以了（见示例 3-10）。

示例 **3-10**　`needles` 的元素在 `haystack` 里出现的次数，两个变量

都是 set 类型

```
found = len(needles & haystack)
```

如果不使用交集操作的话，代码可能就变成了示例 3-11 里那样。

示例 **3-11**　needles 的元素在 haystack 里出现的次数（作用和示例 3-10 中的相同）

```
found = 0
for n in needles:
    if n in haystack:
        found += 1
```

示例 3-10 比示例 3-11 的速度要快一些；另一方面，示例 3-11 可以用在任何可迭代对象 needles 和 haystack 上，而示例 3-10 则要求两个对象都是集合。话再说回来，就算手头没有集合，我们也可以随时建立集合，如示例 3-12 所示。

示例 **3-12**　needles 的元素在 haystack 里出现的次数，这次的代码可以用在任何可迭代对象上

```
found = len(set(needles) & set(haystack))

# 另一种写法：
found = len(set(needles).intersection(haystack))
```

示例 3-12 里的这种写法会牵扯到把对象转化为集合的成本，不过如果 needles 或者是 haystack 中任意一个对象已经是集合，那么示例 3-12 的方案可能就比示例 3-11 里的要更高效。

以上的所有例子的运行时间都能在 3 毫秒左右，在含有 10 000 000 个元素的 haystack 里搜索 1000 个值，算下来大概是每个元素 3 微秒。

除了速度极快的查找功能（这也得归功于它背后的散列表），内置的 set 和 frozenset 提供了丰富的功能和操作，不但让创建集合的方式

丰富多彩，而且对于 set 来讲，我们还可以对集合里已有的元素进行修改。在讨论这些操作之前，先来看一下相关的句法。

3.8.1　集合字面量

除空集之外，集合的字面量——{1}、{1, 2}，等等——看起来跟它的数学形式一模一样。如果是空集，那么必须写成 set() 的形式。



句法的陷阱

不要忘了，如果要创建一个空集，你必须用不带任何参数的构造方法 set()。如果只是写成 {} 的形式，跟以前一样，你创建的其实是个空字典。

在 Python 3 里面，除了空集，集合的字符串表示形式总是以 {...} 的形式出现。

```
>>> s = {1}
>>> type(s)
<class 'set'>
>>> s
{1}
>>> s.pop()
1
>>> s
set()
```

像 {1, 2, 3} 这种字面量句法相比于构造方法（set([1, 2, 3])）要更快且更易读。后者的速度要慢一些，因为 Python 必须先从 set 这个名字来查询构造方法，然后新建一个列表，最后再把这个列表传入到构造方法里。但是如果是像 {1, 2, 3} 这样的字面量，Python 会利用一个专门的叫作 BUILD_SET 的字节码来创建集合。

用 **dis.dis**（反汇编函数）来看看两个方法的字节码的不同：

```
>>> from dis import dis
>>> dis('{1}')                                  ❶
  1           0 LOAD_CONST               0 (1)
```

```
              3 BUILD_SET                1        ❷
              6 RETURN_VALUE
>>> dis('set([1])')                               ❸
  1           0 LOAD_NAME                0 (set)  ❹
              3 LOAD_CONST               0 (1)
              6 BUILD_LIST               1
              9 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             12 RETURN_VALUE
```

❶ 检查 {1} 字面量背后的字节码。

❷ 特殊的字节码 BUILD_SET 几乎完成了所有的工作。

❸ set([1]) 的字节码。

❹ 3 种不同的操作代替了上面的 BUILD_SET：LOAD_NAME、BUILD_LIST 和 CALL_FUNCTION。

由于 Python 里没有针对 frozenset 的特殊字面量句法，我们只能采用构造方法。Python 3 里 frozenset 的标准字符串表示形式看起来就像构造方法调用一样。来看这段控制台对话：

```
>>> frozenset(range(10))
frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9})
```

既然提到了句法，就不得不提一下我们已经熟悉的列表推导，因为也有类似的方式来新建集合。

3.8.2　集合推导

Python 2.7 带来了集合推导（setcomps）和之前在 3.2 节里讲到过的字典推导。示例 3-13 是个简单的例子。

示例 **3-13**　新建一个 Latin-1 字符集合，该集合里的每个字符的 Unicode 名字里都有“SIGN”这个单词

```
>>> from unicodedata import name  ❶
>>> {chr(i) for i in range(32, 256) if 'SIGN' in name(chr(i),'')}  ❷
{'§', '=', '¢', '#', '¤', '<', '¥', 'µ', '×', '$', '¶', '£', '©',
```

```
'°', '+', '÷', '±', '>', '¬', '®', '%'}
```

❶ 从 unicodedata 模块里导入 name 函数，用以获取字符的名字。

❷ 把编码在 32~255 之间的字符的名字里有“SIGN”单词的挑出来，放到一个集合里。

跟句法相关的内容就讲到这里，下面看看用于集合类型的丰富操作。

3.8.3　集合的操作

图 3-2 列出了可变和不可变集合所拥有的方法的概况，其中不少是运算符重载的特殊方法。表 3-2 则包含了数学里集合的各种操作在 Python 中所对应的运算符和方法。其中有些运算符和方法会对集合做就地修改（像 &=、difference_update，等等），这类操作在纯粹的数学世界里是没有意义的，另外 frozenset 也不会实现这些操作。

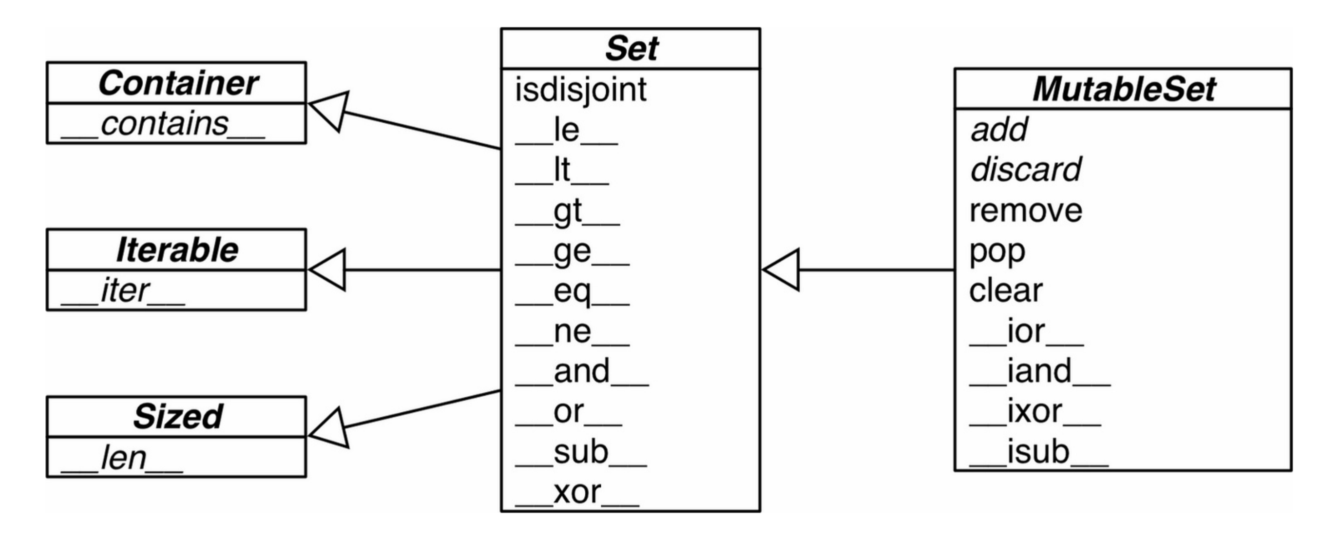


图 **3-2**：**collections.abc** 中，**MutableSet** 和它的超类的 **UML** 类图（箭头从子类指向超类，抽象类和抽象方法的名称以斜体显示，其中省略了反向运算符方法）

表 3-2 中的中缀运算符需要两侧的被操作对象都是集合类型，但是其他的所有方法则只要求所传入的参数是可迭代对象。例如，想求 4 个聚合类型 a、b、c 和 d 的合集，可以用 a.union(b, c, d)，这里 a 必须是个 set，但是 b、c 和 d 则可以是任何类型

的可迭代对象。

表**3-2**：集合的数学运算：这些方法或者会生成新集合，或者会在条件允许的情况下就地修改集合

<table>
<tr><th>数学符号</th><th>Python 运算符</th><th>方法</th><th>描述</th></tr>
<tr><td rowspan="5">S ∩ Z</td><td>s & z</td><td>s.__and__(z)</td><td>s 和 z 的交集</td></tr>
<tr><td rowspan="2">z & s</td><td>s.__rand__(z)</td><td>反向 & 操作</td></tr>
<tr><td>s.intersection(it, ...)</td><td>把可迭代的 it 和其他所有参数转化为集合，然后求它们与 s 的交集</td></tr>
<tr><td rowspan="2">s &= z</td><td>s.__iand__(z)</td><td>把 s 更新为 s 和 z 的交集</td></tr>
<tr><td>s.intersection_update(it, ...)</td><td>把可迭代的 it 和其他所有参数转化为集合，然后求得它们与 s 的交集，然后把 s 更新成这个交集</td></tr>
<tr><td rowspan="5">S ∪ Z</td><td>s | z</td><td>s.__or__(z)</td><td>s 和 z 的并集</td></tr>
<tr><td rowspan="2">z | s</td><td>s.__ror__(z)</td><td>| 的反向操作</td></tr>
<tr><td>s.union(it, ...)</td><td>把可迭代的 it 和其他所有参数转化为集合，然后求它们和 s 的并集</td></tr>
<tr><td rowspan="2">s |= z</td><td>s.__ior__(z)</td><td>把 s 更新为 s 和 z 的并集</td></tr>
<tr><td>s.update(it, ...)</td><td>把可迭代的 it 和其他所有参数转化为集合，然后求它们和 s 的并集，并把 s 更新成这个并集</td></tr>
<tr><td rowspan="5">S \ Z</td><td>s - z</td><td>s.__sub__(z)</td><td>s 和 z 的差集，或者叫作相对补集</td></tr>
<tr><td rowspan="2">z - s</td><td>s.__rsub__(z)</td><td>- 的反向操作</td></tr>
<tr><td>s.difference(it, ...)</td><td>把可迭代的 it 和其他所有参数转化为集合，然后求它们和 s 的差集</td></tr>
<tr><td rowspan="2">s -= z</td><td>s.__isub__(z)</td><td>把 s 更新为它与 z 的差集</td></tr>
<tr><td>s.difference_update(it, ...)</td><td>把可迭代的 it 和其他所有参数转化为集合，求它们和 s 的差集，然后把 s 更新成这个差集</td></tr>
</table>

		s.symmetric_difference(it)	求 s 和 set(it) 的对称差集
S △ Z	s ^ z	s.__xor__(z)	求 s 和 z 的对称差集
	z ^ s	s.__rxor__(z)	^ 的反向操作
		s.symmetric_difference_update(it, …)	把可迭代的 it 和其他所有参数转化为集合，然后求它们和 s 的对称差集，最后把 s 更新成该结果
	s ^= z	s.__ixor__(z)	把 s 更新成它与 z 的对称差集

在写这本书的时候，Python 有个缺陷（issue 8743，http://bugs.python.org/issue8743），里面说到 `set()` 的运算符（`or`、`and`、`sub`、`xor` 和它们相对应的就地修改运算符）要求参数必须是 `set()` 的实例，这就导致这些运算符不能被用在 `collections.abc.Set` 这个子类上面。这个缺陷已经在 Python 2.7 和 Python 3.4 里修复了，在你看到这本书的时候，它已经成了历史。

表 3-3 里列出了返回值是 `True` 和 `False` 的方法和运算符。

表**3-3**：集合的比较运算符，返回值是布尔类型

数学符号	**Python** 运算符	方法	描述
		s.isdisjoint(z)	查看 s 和 z 是否不相交（没有共同元素）
e ∈ S	e in s	s.__contains__(e)	元素 e 是否属于 s
S ⊆ Z	s <= z	s.__le__(z)	s 是否为 z 的子集
		s.issubset(it)	把可迭代的 it 转化为集合，然后查看 s 是否为它的子集
S ⊂ Z	s < z	s.__lt__(z)	s 是否为 z 的真子集
S ⊇ Z	s >= z	s.__ge__(z)	s 是否为 z 的父集
		s.issuperset(it)	把可迭代的 it 转化为集合，然后查看 s 是否为它的父集
S ⊃ Z	s > z	s.__gt__(z)	s 是否为 z 的真父集

除了跟数学上的集合计算有关的方法和运算符，集合类型还有一些为了实用性而添加的方法，其汇总见于表 3-4。

表**3-4**：集合类型的其他方法

	set	frozenset	
s.add(e)	•		把元素 e 添加到 s 中
s.clear()	•		移除掉 s 中的所有元素
s.copy()	•	•	对 s 浅复制
s.discard(e)	•		如果 s 里有 e 这个元素的话，把它移除
s.__iter__()	•	•	返回 s 的迭代器
s.__len__()	•	•	len(s)
s.pop()	•		从 s 中移除一个元素并返回它的值，若 s 为空，则抛出 KeyError 异常
s.remove(e)	•		从 s 中移除 e 元素，若 e 元素不存在，则抛出 KeyError 异常

到这里，我们差不多把集合类型的特性总结完了。

下面会继续探讨字典和集合类型背后的实现，看看它们是如何借助散列表来实现这些功能的。读完这章余下的内容后，就算再遇到 `dict`、`set` 或是其他这一类型的一些莫名其妙的表现，你也不会手足无措。

3.9　dict和set的背后

想要理解 Python 里字典和集合类型的长处和弱点，它们背后的散列表是绕不开的一环。

这一节将会回答以下几个问题。

- Python 里的 `dict` 和 `set` 的效率有多高？
- 为什么它们是无序的？
- 为什么并不是所有的 Python 对象都可以当作 `dict` 的键或 `set` 里的元素？
- 为什么 `dict` 的键和 `set` 元素的顺序是跟据它们被添加的次序而定的，以及为什么在映射对象的生命周期中，这个顺序并不是一成不变的？
- 为什么不应该在迭代循环 `dict` 或是 `set` 的同时往里添加元素？

为了让你有动力研究散列表，下面先来看一个关于 `dict` 和 `set` 效率的实验，实验对象里大概有上百万个元素，而实验结果可能会出乎你的意料。

3.9.1　一个关于效率的实验

所有的 Python 程序员都从经验中得出结论，认为字典和集合的速度是非常快的。接下来我们要通过可控的实验来证实这一点。

为了对比容器的大小对 `dict`、`set` 或 `list` 的 `in` 运算符效率的影响，我创建了一个有 1000 万个双精度浮点数的数组，名叫 `haystack`。另外还有一个包含了 1000 个浮点数的 `needles` 数组，其中 500 个数字是从 `haystack` 里挑出来的，另外 500 个肯定不在 `haystack` 里。

作为 `dict` 测试的基准，我用 `dict.fromkeys()` 来建立了一个含有

1000 个浮点数的名叫 haystack 的字典，并用 timeit 模块测试示例 3-14（与示例 3-11 相同）里这段代码运行所需要的时间。

示例 **3-14** 在 haystack 里查找 needles 的元素，并计算找到的元素的个数

```
found = 0
for n in needles:
    if n in haystack:
        found += 1
```

然后这段基准测试重复了 4 次，每次都把 haystack 的大小变成了上一次的 10 倍，直到里面有 1000 万个元素。最后这些测试的结果列在了表 3-5 中。

表**3-5**：用**in**运算符在**5**个不同大小的**haystack**字典里搜索**1000**个元素所需要的时间。代码运行在一个**Core i7**笔记本上，**Python**版本是**3.4.0**（测试计算的是示例**3-14**里循环的运行时间）

haystack的长度	增长系数	**dict**花费时间	增长系数
1000	1×	0.000202s	1.00×
10 000	10×	0.000140s	0.69×
100 000	100×	0.000228s	1.13×
1 000 000	1000×	0.000290s	1.44×
10 000 000	10 000×	0.000337s	1.67×

也就是说，在我的笔记本上从 1000 个字典键里搜索 1000 个浮点数所需的时间是 0.000202 秒，把同样的搜索在含有 10 000 000 个元素的字典

里进行一遍，只需要 0.000337 秒。换句话说，在一个有 1000 万个键的字典里查找 1000 个数，花在每个数上的时间不过是 0.337 微秒——没错，相当于平均每个数差不多三分之一微秒。

作为对比，我把 haystack 换成了 set 和 list 类型，重复了同样的增长大小的实验。对于 set，除了上面的那个循环的运行时间，我还测量了示例 3-15 那行代码，这段代码也计算了 needles 中出现在 haystack 中的元素的个数。

示例 **3-15**　利用交集来计算 needles 中出现在 haystack 中的元素的个数

```
found = len(needles & haystack)
```

表 3-6 列出了所有测试的结果。最快的时间来自“集合交集花费时间”这一列，这一列的结果是示例 3-15 中利用集合 & 操作的代码的效果。不出所料的是，最糟糕的表现来自“列表花费时间”这一列。由于列表的背后没有散列表来支持 in 运算符，每次搜索都需要扫描一次完整的列表，导致所需的时间跟据 haystack 的大小呈线性增长。

表**3-6**：在**5**个不同大小的**haystack**里搜索**1000**个元素所需的时间，**haystack**分别以字典、集合和列表的形式出现。测试环境是一个有**Core i7**处理器的笔记本，**Python**版本是**3.4.0**（测试所测量的代码是示例**3-14**中的循环和示例**3-15**的集合**&**操作）

haystack的长度	增长系数	dict花费时间	增长系数	集合花费时间	增长系数	集合交集花费时间	增长系数	列表花费时间	增长系数
1000	1×	0.000202s	1.00×	0.000143s	1.00×	0.000087s	1.00×	0.010556s	1.00×
10 000	10×	0.000140s	0.69×	0.000147s	1.03×	0.000092s	1.06×	0.086586s	8.20×
100 000	100×	0.000228s	1.13×	0.000241s	1.69×	0.000163s	1.87×	0.871560s	82.57×

1 000 000	1000×	0.000290s	1.44×	0.000332s	2.32×	0.000250s	2.87×	9.189616s	870.56×
10 000 000	10 000×	0.000337s	1.67×	0.000387s	2.71×	0.000314s	3.61×	97.948056s	9278.90×

如果在你的程序里有任何的磁盘输入 / 输出，那么不管查询有多少个元素的字典或集合，所耗费的时间都能忽略不计（前提是字典或者集合不超过内存大小）。可以仔细看看跟表 3-6 有关的代码，另外在附录 A 的示例 A-1 中还有相关的讨论。

把字典和集合的运行速度之快的事实抓在手里之后，让我们来看看它背后的原因。对散列表内部结构的讨论，能解释诸如为什么键是无序且不稳定的。

3.9.2　字典中的散列表

这一节笼统地描述了 Python 如何用散列表来实现 `dict` 类型，有些细节只是一笔带过，像 CPython 里的一些优化技巧 [8] 就没有提到。但是总体来说描述是准确的。

> [8]Python 源码 `dictobject.c` 模块（http://hg.python.org/cpython/file/tip/Objects/dictobject.c）里有丰富的注释，另外延伸阅读中有对《代码之美》一书的引用。

为了简单起见，这里先集中讨论 `dict` 的内部结构，然后再延伸到集合上面。

散列表其实是一个稀疏数组（总是有空白元素的数组称为稀疏数组）。在一般的数据结构教材中，散列表里的单元通常叫作表元（bucket）。在 `dict` 的散列表当中，每个键值对都占用一个表元，每个表元都有两个部分，一个是对键的引用，另一个是对值的引用。因为所有表元的大小一致，所以可以通过偏移量来读取某个表元。

因为 Python 会设法保证大概还有三分之一的表元是空的，所以在快要达到这个阈值的时候，原有的散列表会被复制到一个更大的空间里面。

如果要把一个对象放入散列表，那么首先要计算这个元素键的散列值。Python 中可以用 `hash()` 方法来做这件事情，接下来会介绍这一点。

01. 散列值和相等性

内置的 `hash()` 方法可以用于所有的内置类型对象。如果是自定义对象调用 `hash()` 的话，实际上运行的是自定义的 `__hash__`。如果两个对象在比较的时候是相等的，那它们的散列值必须相等，否则散列表就不能正常运行了。例如，如果 `1 == 1.0` 为真，那么 `hash(1) == hash(1.0)` 也必须为真，但其实这两个数字（整型和浮点）的内部结构是完全不一样的。[9]

为了让散列值能够胜任散列表索引这一角色，它们必须在索引空间中尽量分散开来。这意味着在最理想的状况下，越是相似但不相等的对象，它们散列值的差别应该越大。示例 3-16 是一段代码输出，这段代码被用来比较散列值的二进制表达的不同。注意其中 1 和 1.0 的散列值是相同的，而 1.0001、1.0002 和 1.0003 的散列值则非常不同。

示例 **3-16** 在32 位的 Python 中，1、1.0001、1.0002 和 1.0003 这几个数的散列值的二进制表达对比（上下两个二进制间不同的位被 ! 高亮出来，表格的最右列显示了有多少位不相同）

```
32-bit Python build
1        00000000000000000000000000000001
                                          != 0
1.0      00000000000000000000000000000001
------------------------------------------------
1.0      00000000000000000000000000000001
           ! !!! ! !! ! !    ! ! !! !!!   != 16
1.0001   00101110101101010000101011011101
------------------------------------------------
1.0001   00101110101101010000101011011101
          !!!  !!!! !!!!!   !!!!! !!  !   != 20
1.0002   01011101011010100001010110111001
------------------------------------------------
1.0002   01011101011010100001010110111001
          ! !   ! !!! ! !  !! ! !  ! !!!! != 17
1.0003   00001100000111110010000010010110
------------------------------------------------
```

用来计算示例 3-16 的程序见于附录 A。尽管程序里大部分代码都是用来整理输出格式的，考虑到完整性，我还是把全部的代码放在示例 A-3 中了。

> 从 Python 3.3 开始，`str`、`bytes` 和 `datetime` 对象的散列值计算过程中多了随机的“加盐”这一步。所加盐值是 Python 进程内的一个常量，但是每次启动 Python 解释器都会生成一个不同的盐值。随机盐值的加入是为了防止 DOS 攻击而采取的一种安全措施。在 `__hash__` 特殊方法的文档（https://docs.python.org/3/reference/datamodel.html#object.__hash__）里有相关的详细信息。

了解对象散列值相关的基本概念之后，我们可以深入到散列表工作原理背后的算法了。

02. 散列表算法

为了获取 `my_dict[search_key]` 背后的值，Python 首先会调用 `hash(search_key)` 来计算 `search_key` 的散列值，把这个值最低的几位数字当作偏移量，在散列表里查找表元（具体取几位，得看当前散列表的大小）。若找到的表元是空的，则抛出 `KeyError` 异常。若不是空的，则表元里会有一对 `found_key:found_value`。这时候 Python 会检验 `search_key == found_key` 是否为真，如果它们相等的话，就会返回 `found_value`。

如果 `search_key` 和 `found_key` 不匹配的话，这种情况称为散列冲突。发生这种情况是因为，散列表所做的其实是把随机的元素映射到只有几位的数字上，而散列表本身的索引又只依赖于这个数字的一部分。为了解决散列冲突，算法会在散列值中另外再取几位，然后用特殊的方法处理一下，把新得到的数字再当作索引来寻找表元。[10] 若这次找到的表元是空的，则同样抛出 `KeyError`；若非空，或者键匹配，则返回这个值；或者又发现了散列冲突，则重复以上的步骤。图 3-3 展示了这个算法的示意图。

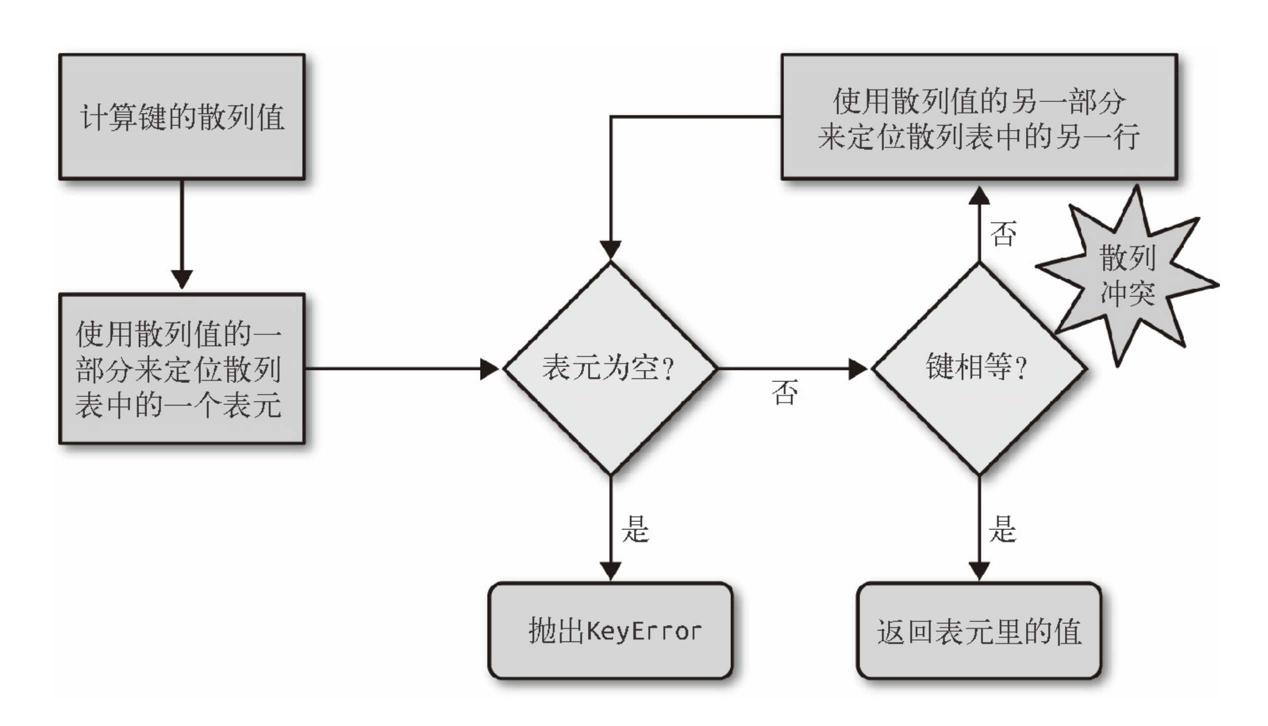


图 **3-3**：从字典中取值的算法流程图；给定一个键，这个算法要么返回一个值，要么抛出 **`KeyError`** 异常

添加新元素和更新现有键值的操作几乎跟上面一样。只不过对于前者，在发现空表元的时候会放入一个新元素；对于后者，在找到相对应的表元后，原表里的值对象会被替换成新值。

另外在插入新值时，Python 可能会按照散列表的拥挤程度来决定是否要重新分配内存为它扩容。如果增加了散列表的大小，那散列值所占的位数和用作索引的位数都会随之增加，这样做的目的是为了减少发生散列冲突的概率。

表面上看，这个算法似乎很费事，而实际上就算 `dict` 里有数百万个元素，多数的搜索过程中并不会有冲突发生，平均下来每次搜索可能会有一到两次冲突。在正常情况下，就算是最不走运的键所遇到的冲突的次数用一只手也能数过来。

了解 `dict` 的工作原理能让我们知道它的所长和所短，以及从它衍生而来的数据类型的优缺点。下面就来看看 `dict` 这些特点背后的原因。

[9]既然提到了整型，CPython 的实现细节里有一条是：如果有一个整型对象，而且它能被存进一个机器字中，那么它的散列值就是它本身的值。

> [10]在散列冲突的情况下，用 C 语言写的用来打乱散列值位的算法的名字很有意思，叫 `perturb`。详见 CPython 源码里的 `dictobject.c`（https://hg.python.org/cpython/file/tip/Objects/dictobject.c）。

3.9.3 `dict`的实现及其导致的结果

下面的内容会讨论使用散列表给 `dict` 带来的优势和限制都有哪些。

01. 键必须是可散列的

    一个可散列的对象必须满足以下要求。

    (1) 支持 `hash()` 函数，并且通过 `__hash__()` 方法所得到的散列值是不变的。

    (2) 支持通过 `__eq__()` 方法来检测相等性。

    (3) 若 `a == b` 为真，则 `hash(a) == hash(b)` 也为真。

    所有由用户自定义的对象默认都是可散列的，因为它们的散列值由 `id()` 来获取，而且它们都是不相等的。

    > 如果你实现了一个类的 `__eq__` 方法，并且希望它是可散列的，那么它一定要有个恰当的 `__hash__` 方法，保证在 `a == b` 为真的情况下 `hash(a) == hash(b)` 也必定为真。否则就会破坏恒定的散列表算法，导致由这些对象所组成的字典和集合完全失去可靠性，这个后果是非常可怕的。另一方面，如果一个含有自定义的 `__eq__` 依赖的类处于可变的状态，那就不要在这个类中实现 `__hash__` 方法，因为它的实例是不可散列的。

02. 字典在内存上的开销巨大

    由于字典使用了散列表，而散列表又必须是稀疏的，这导致它在空间上的效率低下。举例而言，如果你需要存放数量巨大的记录，那么放在由元组或是具名元组构成的列表中会是比较好的选择；最好

不要根据 JSON 的风格，用由字典组成的列表来存放这些记录。用元组取代字典就能节省空间的原因有两个：其一是避免了散列表所耗费的空间，其二是无需把记录中字段的名字在每个元素里都存一遍。

在用户自定义的类型中，`__slots__` 属性可以改变实例属性的存储方式，由 `dict` 变成 `tuple`，相关细节在 9.8 节会谈到。

记住我们现在讨论的是空间优化。如果你手头有几百万个对象，而你的机器有几个 GB 的内存，那么空间的优化工作可以等到真正需要的时候再开始计划，因为优化往往是可维护性的对立面。

03. 键查询很快

`dict` 的实现是典型的空间换时间：字典类型有着巨大的内存开销，但它们提供了无视数据量大小的快速访问——只要字典能被装在内存里。正如表 3-5 所示，如果把字典的大小从 1000 个元素增加到 10 000 000 个，查询时间也不过是原来的 2.8 倍，从 0.000163 秒增加到了 0.00456 秒。这意味着在一个有 1000 万个元素的字典里，每秒能进行 200 万个键查询。

04. 键的次序取决于添加顺序

当往 `dict` 里添加新键而又发生散列冲突的时候，新键可能会被安排存放到另一个位置。于是下面这种情况就会发生：由 `dict([key1, value1), (key2, value2)]` 和 `dict([key2, value2], [key1, value1])` 得到的两个字典，在进行比较的时候，它们是相等的；但是如果在 `key1` 和 `key2` 被添加到字典里的过程中有冲突发生的话，这两个键出现在字典里的顺序是不一样的。

示例 3-17 展示了这个现象。这个示例用同样的数据创建了 3 个字典，唯一的区别就是数据出现的顺序不一样。可以看到，虽然键的次序是乱的，这 3 个字典仍然被视作相等的。

示例 **3-17** dialcodes.py 将同样的数据以不同的顺序添加到 3 个字典里

```
# 世界人口数量前10位国家的电话区号
DIAL_CODES = [
        (86, 'China'),
        (91, 'India'),
        (1, 'United States'),
        (62, 'Indonesia'),
        (55, 'Brazil'),
        (92, 'Pakistan'),
        (880, 'Bangladesh'),
        (234, 'Nigeria'),
        (7, 'Russia'),
        (81, 'Japan'),
    ]

d1 = dict(DIAL_CODES)  ❶
print('d1:', d1.keys())
d2 = dict(sorted(DIAL_CODES))  ❷
print('d2:', d2.keys())
d3 = dict(sorted(DIAL_CODES, key=lambda x:x[1]))  ❸
print('d3:', d3.keys())
assert d1 == d2 and d2 == d3  ❹
```

❶ 创建 d1 的时候，数据元组的顺序是按照国家的人口排名来决定的。

❷ 创建 d2 的时候，数据元组的顺序是按照国家的电话区号来决定的。

❸ 创建 d3 的时候，数据元组的顺序是按照国家名字的英文拼写来决定的。

❹ 这些字典是相等的，因为它们所包含的数据是一样的。示例 3-18 里是上面例子的输出。

示例 **3-18**　dialcodes.py 的输出中，3 个字典的键的顺序是不一样的

```
d1: dict_keys([880, 1, 86, 55, 7, 234, 91, 92, 62, 81])
d2: dict_keys([880, 1, 91, 86, 81, 55, 234, 7, 92, 62])
d3: dict_keys([880, 81, 1, 86, 55, 7, 234, 91, 92, 62])
```

05. 往字典里添加新键可能会改变已有键的顺序

无论何时往字典里添加新的键，Python 解释器都可能做出为字典扩容的决定。扩容导致的结果就是要新建一个更大的散列表，并把字典里已有的元素添加到新表里。这个过程中可能会发生新的散列冲突，导致新散列表中键的次序变化。要注意的是，上面提到的这些变化是否会发生以及如何发生，都依赖于字典背后的具体实现，因此你不能很自信地说自己知道背后发生了什么。如果你在迭代一个字典的所有键的过程中同时对字典进行修改，那么这个循环很有可能会跳过一些键——甚至是跳过那些字典中已经有的键。

由此可知，不要对字典同时进行迭代和修改。如果想扫描并修改一个字典，最好分成两步来进行：首先对字典迭代，以得出需要添加的内容，把这些内容放在一个新字典里；迭代结束之后再对原有字典进行更新。

> 在 Python 3 中，`.keys()`、`.items()` 和 `.values()` 方法返回的都是字典视图。也就是说，这些方法返回的值更像集合，而不是像 Python 2 那样返回列表。视图还有动态的特性，它们可以实时反馈字典的变化。

现在已经可以把学到的有关散列表的知识应用在集合上面了。

3.9.4 `set`的实现以及导致的结果

`set` 和 `frozenset` 的实现也依赖散列表，但在它们的散列表里存放的只有元素的引用（就像在字典里只存放键而没有相应的值）。在 `set` 加入到 Python 之前，我们都是把字典加上无意义的值当作集合来用的。

在 3.9.3 节中所提到的字典和散列表的几个特点，对集合来说几乎都是适用的。为了避免太多重复的内容，这些特点总结如下。

- 集合里的元素必须是可散列的。
- 集合很消耗内存。
- 可以很高效地判断元素是否存在于某个集合。

- 元素的次序取决于被添加到集合里的次序。
- 往集合里添加元素，可能会改变集合里已有元素的次序。

3.10　本章小结

字典算得上是 Python 的基石。除了基本的 dict 之外，标准库还提供现成且好用的特殊映射类型，比如 defaultdict、OrderedDict、ChainMap 和 Counter。这些映射类型都属于 collections 模块，这个模块还提供了便于扩展的 UserDict 类。

大多数映射类型都提供了两个很强大的方法：setdefault 和 update。setdefault 方法可以用来更新字典里存放的可变值（比如列表），从而避免了重复的键搜索。update 方法则让批量更新成为可能，它可以用来插入新值或者更新已有键值对，它的参数可以是包含 (key, value) 这种键值对的可迭代对象，或者关键字参数。映射类型的构造方法也会利用 update 方法来让用户可以使用别的映射对象、可迭代对象或者关键字参数来创建新对象。

在映射类型的 API 中，有个很好用的方法是 __missing__，当对象找不到某个键的时候，可以通过这个方法自定义会发生什么。

collections.abc 模块提供了 Mapping 和 MutableMapping 这两个抽象基类，利用它们，我们可以进行类型查询或者引用。不太为人所知的 MappingProxyType 可以用来创建不可变映射对象，它被封装在 types 模块中。另外还有 Set 和 MutableSet 这两个抽象基类。

dict 和 set 背后的散列表效率很高，对它的了解越深入，就越能理解为什么被保存的元素会呈现出不同的顺序，以及已有的元素顺序会发生变化的原因。同时，速度是以牺牲空间为代价而换来的。

3.11　延伸阅读

Python 标准库中的“8.3. collections—Container datatypes”一节（https://docs.python.org/3/library/collections.html）提到了关于一些映射类型的例子和使用技巧。如果想要创建新的映射类型，或者是体会一下现有的映射类型的实现方式，Python 模块 Lib/collections/`__init__`.py 的源码是一个很好的参考。

《Python Cookbook（第 3 版）中文版》（David Beazley 和 Brian K. Jones 著）的第 1 章中有 20 个关于数据结构的使用技巧，大多数都在讲 `dict` 的巧妙用法。

“Python 的字典类：如何打造全能战士”是《代码之美》第 18 章的标题，这一章集中解释了 Python 字典背后的工作原理。A.M. Kuchling 是这一章的作者，同时他还是 Python 的核心开发者，并撰写了很多 Python 的官方文档和指南。同时 CPython 模块里的 `dictobject.c` 源文件（https://hg.python.org/cpython/file/tip/Objects/dictobject.c）还提供了大量的注释。Brandon Craig Rhodes 的讲座“The Mighty Dictionary”（http://pyvideo.org/video/276/the-mighty-dictionary-55）对散列表做了很精彩的讲解，有趣的是他的幻灯片里也包含了大量的表格。

关于为什么要在语言里加入集合这种数据类型，当初也是有一番考量的。具体情况在“PEP 218 — Adding a Built-In Set Object Type”（https://www.python.org/dev/peps/pep-0218/）中有所记录。在 PEP 128 刚刚通过的时候，还没有针对 `set` 的特殊字面量句法。后来 Python 3 里加入了对 `set` 字面量句法的支持，然后这个实现又被向后兼容到了 Python 2.7 里，同时被移植的还有 `dict` 和 `set` 推导。“PEP 274 — Dict Comprehensions”（https://www.python.org/dev/peps/pep-0274/）就是字典推导的出生证；然而我找不到任何关于集合推导的 PEP，当然很有可能是因为这两个功能太接近了。

杂谈

我的朋友 Geraldo Cohen 曾经说过，Python 的特点是“简单而正确”。

dict 类型正是这一特点的完美体现——对它的优化只为一个目标：更好地实现对随机键的读取。而优化的结果非常好，由于速度快而且够健壮，它大量地应用于 Python 的解释器当中。如果对排序有要求，那么还可以选择OrderedDict。然而对于映射类型来说，保持元素的顺序并不是一个常用需求，因此会把它排除在核心功能之外，而以标准库的形式提供其他衍生的类型。

与之形成鲜明对比的是 PHP。在 PHP 手册中，数组的描述如下（http://php.net/manual/en/language.types.array.php）：

> PHP 中的数组实际上是一个有序的映射——映射类型存放的是键值对。这个映射类型被优化为可充当不同的角色。它可以当作数组、列表（向量）、散列表（映射类型的一种实现）、字典、集合类型、栈、队列或其他可能的数据类型。

单凭这段话，我无法想象 PHP 把 list 和 OrderedDict 混合实现的成本有多大。

本书前两章的目的是展示 Python 中的集合类型为特定的使用场景做了怎样的优化。我特意强调了在 list 和 dict 的常规用法之外还有那些特殊的使用情景。

在遇到 Python 之前，我主要使用 Perl、PHP 和 JavaScript 做网站开发。我很喜欢这些语言中跟映射类型相关的字面量句法特性。某些时候我不得不使用 Java 和 C，然后我就会疯狂地想念这些特性。好用的映射类型的字面量句法可以帮助开发者轻松实现配置和表格相关的开发，也能让我们很方便地为原型开发或者测试准备好数据容器。 Java 由于没有这个特性，不得不用复杂且冗长的 XML 来替代。

JSON 被当作“瘦身版 XML”（http://www.json.org/fatfree.html）。在很多情景下，JSON 都成功取代了 XML。由于拥有紧凑的列表和字典表达，JSON 格式可以完美地用于数据交换。

PHP 和 Ruby 的散列语法借鉴了 Perl，它们都用 => 作为键和值的连接。JavaScript 则从 Python 那儿偷师，使用了 :。而 JSON 又从 JavaScript 发展而来，它的语法正好是 Python 句法的子集。因此，除了在 true、false 和 null 这几个值的拼写上有出入之外，

JSON 和 Python 是完全兼容的。于是，现在大家用来交换数据的格式全是 Python 的 `dict` 和 `list`。

简单而正确。

第 4 章　文本和字节序列

> 人类使用文本，计算机使用字节序列。[1]
>
> ——Esther Nam 和 Travis Fischer
> “Character Encoding and Unicode in Python”

> [1]PyCon 2014，“Character Encoding and Unicode in Python”演讲的第 12 张幻灯片 [幻灯片（http://www.slideshare.net/fischertrav/character-encoding-unicode-how-to-with-dignity-33352863），视频（http://pyvideo.org/pycon-us-2014/character-encoding-and-unicode-in-python.html）]。

Python 3 明确区分了人类可读的文本字符串和原始的字节序列。隐式地把字节序列转换成 Unicode 文本已成过去。本章将要讨论 Unicode 字符串、二进制序列，以及在二者之间转换时使用的编码。

深入理解 Unicode 对你可能十分重要，也可能无关紧要，这取决于 Python 编程的场景。说到底，本章涵盖的问题对只处理 ASCII 文本的程序员没有影响。但是即便如此，也不能避而不谈字符串和字节序列的区别。此外，你会发现专门的二进制序列类型所提供的功能，有些是 Python 2 中“全功能”的 `str` 类型不具有的。

本章将讨论下述话题：

- 字符、码位和字节表述
- `bytes`、`bytearray` 和 `memoryview` 等二进制序列的独特特性
- 全部 Unicode 和陈旧字符集的编解码器
- 避免和处理编码错误
- 处理文本文件的最佳实践
- 默认编码的陷阱和标准 I/O 的问题
- 规范化 Unicode 文本，进行安全的比较

- 规范化、大小写折叠和暴力移除音调符号的实用函数
- 使用 `locale` 模块和 PyUCA 库正确地排序 Unicode 文本
- Unicode 数据库中的字符元数据
- 能处理字符串和字节序列的双模式 API

接下来先从字符、码位和字节序列开始。

4.1　字符问题

“字符串”是个相当简单的概念：一个字符串是一个字符序列。问题出在“字符”的定义上。

在 2015 年，“字符”的最佳定义是 Unicode 字符。因此，从 Python 3 的 `str` 对象中获取的元素是 Unicode 字符，这相当于从 Python 2 的 `unicode` 对象中获取的元素，而不是从 Python 2 的 `str` 对象中获取的原始字节序列。

Unicode 标准把字符的标识和具体的字节表述进行了如下的明确区分。

- 字符的标识，即码位，是 0~1 114 111 的数字（十进制），在 Unicode 标准中以 4~6 个十六进制数字表示，而且加前缀“U+”。例如，字母 A 的码位是 U+0041，欧元符号的码位是 U+20AC，高音谱号的码位是 U+1D11E。在 Unicode 6.3 中（这是 Python 3.4 使用的标准），约 10% 的有效码位有对应的字符。

- 字符的具体表述取决于所用的编码。编码是在码位和字节序列之间转换时使用的算法。在 UTF-8 编码中，A（U+0041）的码位编码成单个字节 `\x41`，而在 UTF-16LE 编码中编码成两个字节 `\x41\x00`。再举个例子，欧元符号（U+20AC）在 UTF-8 编码中是三个字节——`\xe2\x82\xac`，而在 UTF-16LE 中编码成两个字节：`\xac\x20`。

把码位转换成字节序列的过程是编码；把字节序列转换成码位的过程是解码。示例 4-1 阐释了这一区分。

示例 **4-1**　编码和解码

```
>>> s = 'café'
>>> len(s)  # ❶
4
>>> b = s.encode('utf8')  # ❷
>>> b
b'caf\xc3\xa9'  # ❸
>>> len(b)  # ❹
```

```
5
>>> b.decode('utf8')  # ❺
'café
```

❶ `'café'` 字符串有 4 个 Unicode 字符。

❷ 使用 UTF-8 把 `str` 对象编码成 `bytes` 对象。

❸ `bytes` 字面量以 `b` 开头。

❹ 字节序列 `b` 有 5 个字节（在 UTF-8 中，“é”的码位编码成两个字节）。

❺ 使用 UTF-8 把 `bytes` 对象解码成 `str` 对象。

> 如果想帮助自己记住 `.decode()` 和 `.encode()` 的区别，可以把字节序列想成晦涩难懂的机器磁芯转储，把 Unicode 字符串想成“人类可读”的文本。那么，把字节序列变成人类可读的文本字符串就是解码，而把字符串变成用于存储或传输的字节序列就是编码。

虽然 Python 3 的 `str` 类型基本相当于 Python 2 的 `unicode` 类型，只不过是换了个新名称，但是 Python 3 的 `bytes` 类型却不是把 `str` 类型换个名称那么简单，而且还有关系紧密的 `bytearray` 类型。因此，在讨论编码和解码的问题之前，有必要先来介绍一下二进制序列类型。

4.2　字节概要

新的二进制序列类型在很多方面与 Python 2 的 `str` 类型不同。首先要知道，Python 内置了两种基本的二进制序列类型：Python 3 引入的不可变 `bytes` 类型和 Python 2.6 添加的可变 `bytearray` 类型。（Python 2.6 也引入了 `bytes` 类型，但那只不过是 `str` 类型的别名，与 Python 3 的 `bytes` 类型不同。）

`bytes` 或 `bytearray` 对象的各个元素是介于 0~255（含）之间的整数，而不像 Python 2 的 `str` 对象那样是单个的字符。然而，二进制序列的切片始终是同一类型的二进制序列，包括长度为 1 的切片，如示例 4-2 所示。

示例 **4-2**　包含 5 个字节的 `bytes` 和 `bytearray` 对象

```
>>> cafe = bytes('café', encoding='utf_8') ❶
>>> cafe
b'caf\xc3\xa9'
>>> cafe[0] ❷
99
>>> cafe[:1] ❸
b'c'
>>> cafe_arr = bytearray(cafe)
>>> cafe_arr ❹
bytearray(b'caf\xc3\xa9')
>>> cafe_arr[-1:] ❺
bytearray(b'\xa9')
```

❶ `bytes` 对象可以从 `str` 对象使用给定的编码构建。

❷ 各个元素是 `range(256)` 内的整数。

❸ `bytes` 对象的切片还是 `bytes` 对象，即使是只有一个字节的切片。

❹ `bytearray` 对象没有字面量句法，而是以 `bytearray()` 和字节序列字面量参数的形式显示。

❺ `bytearray` 对象的切片还是 `bytearray` 对象。

my_bytes[0] 获取的是一个整数，而 my_bytes[:1] 返回的是一个长度为 1 的 bytes 对象——这一点应该不会让人意外。s[0] == s[:1] 只对 str 这个序列类型成立。不过，str 类型的这个行为十分罕见。对其他各个序列类型来说，s[i] 返回一个元素，而 s[i:i+1] 返回一个相同类型的序列，里面是 s[i] 元素。

虽然二进制序列其实是整数序列，但是它们的字面量表示法表明其中有 ASCII 文本。因此，各个字节的值可能会使用下列三种不同的方式显示。

- 可打印的 ASCII 范围内的字节（从空格到 ~），使用 ASCII 字符本身。
- 制表符、换行符、回车符和 \ 对应的字节，使用转义序列 \t、\n、\r 和 \\。
- 其他字节的值，使用十六进制转义序列（例如，\x00 是空字节）。

因此，在示例 4-2 中，我们看到的是 b'caf\xc3\xa9'：前 3 个字节 b'caf' 在可打印的 ASCII 范围内，后两个字节则不然。

除了格式化方法（format 和 format_map）和几个处理 Unicode 数据的方法（包括 casefold、isdecimal、isidentifier、isnumeric、isprintable 和 encode）之外，str 类型的其他方法都支持 bytes 和 bytearray 类型。这意味着，我们可以使用熟悉的字符串方法处理二进制序列，如 endswith、replace、strip、translate、upper 等，只有少数几个其他方法的参数是 bytes 对象，而不是 str 对象。此外，如果正则表达式编译自二进制序列而不是字符串，re 模块中的正则表达式函数也能处理二进制序列。Python 3.0~3.4 不能使用 % 运算符处理二进制序列，但是根据“PEP 461—Adding % formatting to bytes and bytearray”（https://www.python.org/dev/peps/pep-0461/），Python 3.5 应该会支持。

二进制序列有个类方法是 str 没有的，名为 fromhex，它的作用是解析十六进制数字对（数字对之间的空格是可选的），构建二进制序列：

```
>>> bytes.fromhex('31 4B CE A9')
b'1K\xce\xa9'
```

构建 bytes 或 bytearray 实例还可以调用各自的构造方法，传入下述参数。

- 一个 str 对象和一个 encoding 关键字参数。
- 一个可迭代对象，提供 0~255 之间的数值。
- 一个整数，使用空字节创建对应长度的二进制序列。[Python 3.5 会把这个构造方法标记为“过时的”，Python 3.6 会将其删除。参见“PEP 467—Minor API improvements for binary sequences”（https://www.python.org/dev/peps/pep-0467/）。]
- 一个实现了缓冲协议的对象（如 bytes、bytearray、memoryview、array.array）；此时，把源对象中的字节序列复制到新建的二进制序列中。

使用缓冲类对象构建二进制序列是一种低层操作，可能涉及类型转换。示例 4-3 做了演示。

示例 **4-3** 使用数组中的原始数据初始化 bytes 对象

```
>>> import array
>>> numbers = array.array('h', [-2, -1, 0, 1, 2]) ❶
>>> octets = bytes(numbers) ❷
>>> octets
b'\xfe\xff\xff\xff\x00\x00\x01\x00\x02\x00' ❸
```

❶ 指定类型代码 h，创建一个短整数（16 位）数组。

❷ octets 保存组成 numbers 的字节序列的副本。

❸ 这些是表示那 5 个短整数的 10 个字节。

使用缓冲类对象创建 bytes 或 bytearray 对象时，始终复制源对象中的字节序列。与之相反，memoryview 对象允许在二进制数据结构之间共享内存。如果想从二进制序列中提取结构化信息，struct 模块是重要的工具。下一节会使用这个模块处理 bytes 和 memoryview 对象。

结构体和内存视图

struct 模块提供了一些函数，把打包的字节序列转换成不同类型字段组成的元组，还有一些函数用于执行反向转换，把元组转换成打包的字节序列。struct 模块能处理 bytes、bytearray 和 memoryview 对象。

如 2.9.2 节所述，memoryview 类不是用于创建或存储字节序列的，而是共享内存，让你访问其他二进制序列、打包的数组和缓冲中的数据切片，而无需复制字节序列，例如 Python Imaging Library（PIL）[2] 就是这样处理图像的。

> [2]Pillow（https://pillow.readthedocs.org/en/latest/）是 PIL 最活跃的派生库。

示例 4-4 展示了如何使用 memoryview 和 struct 提取一个 GIF 图像的宽度和高度。

示例 **4-4**　使用 memoryview 和 struct 查看一个 GIF 图像的首部

```
>>> import struct
>>> fmt = '<3s3sHH'  # ❶
>>> with open('filter.gif', 'rb') as fp:
...     img = memoryview(fp.read())  # ❷
...
>>> header = img[:10]  # ❸
>>> bytes(header)  # ❹
b'GIF89a+\x02\xe6\x00'
>>> struct.unpack(fmt, header)  # ❺
(b'GIF', b'89a', 555, 230)
>>> del header  # ❻
>>> del img
```

❶ 结构体的格式：< 是小字节序，3s3s 是两个 3 字节序列，HH 是两个 16 位二进制整数。

❷ 使用内存中的文件内容创建一个 `memoryview` 对象……

❸ ……然后使用它的切片再创建一个 `memoryview` 对象；这里不会复制字节序列。

❹ 转换成字节序列，这只是为了显示；这里复制了 10 字节。

❺ 拆包 `memoryview` 对象，得到一个元组，包含类型、版本、宽度和高度。

❻ 删除引用，释放 `memoryview` 实例所占的内存。

注意，`memoryview` 对象的切片是一个新 `memoryview` 对象，而且不会复制字节序列。[本书的技术审校之一 Leonardo Rochael 指出，如果使用 `mmap` 模块把图像打开为内存映射文件，那么会复制少量字节。本书不会讨论 `mmap`，如果你经常读取和修改二进制文件，可以阅读“mmap—Memory-mapped file support”（https://docs.python.org/3/library/mmap.html）来进一步学习。]

本书不会深入介绍 `memoryview` 和 `struct` 模块，如果要处理二进制数据，可以阅读它们的文档：“Built-in Types » Memory Views”（https://docs.python.org/3/library/stdtypes.html#memory-views）和“struct—Interpret bytes as packed binary data”（https://docs.python.org/3/library/struct.html）。

简要探讨 Python 的二进制序列类型之后，下面说明如何在它们和字符串之间转换。

4.3　基本的编解码器

Python 自带了超过 100 种编解码器（codec, encoder/decoder），用于在文本和字节之间相互转换。每个编解码器都有一个名称，如 'utf_8'，而且经常有几个别名，如 'utf8'、'utf-8' 和 'U8'。这些名称可以传给 open()、str.encode()、bytes.decode() 等函数的 encoding 参数。示例 4-5 使用 3 个编解码器把相同的文本编码成不同的字节序列。

示例 **4-5**　使用 3 个编解码器编码字符串“El Niño”，得到的字节序列差异很大

```
>>> for codec in ['latin_1', 'utf_8', 'utf_16']:
...     print(codec, 'El Niño'.encode(codec), sep='\t')
...
latin_1 b'El Ni\xf1o'
utf_8   b'El Ni\xc3\xb1o'
utf_16  b'\xff\xfeE\x00l\x00 \x00N\x00i\x00\xf1\x00o\x00'
```

图 4-1 展示了不同编解码器对“A”和高音谱号等字符编码后得到的字节序列。注意，后 3 种是可变长度的多字节编码。

char.	code point	ascii	latin1	cp1252	cp437	gb2312	utf-8	utf-16le
A	U+0041	41	41	41	41	41	41	41 00
¿	U+00BF	*	BF	BF	A8	*	C2 BF	BF 00
Ã	U+00C3	*	C3	C3	*	*	C3 83	C3 00
á	U+00E1	*	E1	E1	A0	A8 A2	C3 A1	E1 00
Ω	U+03A9	*	*	*	EA	A6 B8	CE A9	A9 03
ڿ	U+06BF	*	*	*	*	*	DA BF	BF 06
“	U+201C	*	*	93	*	A1 B0	E2 80 9C	1C 20
€	U+20AC	*	*	80	*	*	E2 82 AC	AC 20
┌	U+250C	*	*	*	DA	A9 B0	E2 94 8C	0C 25
气	U+6C14	*	*	*	*	C6 F8	E6 B0 94	14 6C
氣	U+6C23	*	*	*	*	*	E6 B0 A3	23 6C
𝄞	U+1D11E	*	*	*	*	*	F0 9D 84 9E	34 D8 1E DD

图 **4-1**：**12**　个字符，它们的码位及不同编码的字节表述（十六进制，星号表明该编码不支持表示该字符）

图 4-1 中的星号表明，某些编码（如 ASCII 和多字节的 GB2312）不能表示所有 Unicode 字符。然而，UTF 编码的设计目的就是处理每一个 Unicode 码位。

图 4-1 中展示的是一些典型编码，介绍如下。

`latin1`（即 `iso8859_1`）

　　一种重要的编码，是其他编码的基础，例如 `cp1252` 和 Unicode（注意，`latin1` 与 `cp1252` 的字节值是一样的，甚至连码位也相同）。

`cp1252`

　　Microsoft 制定的 `latin1` 超集，添加了有用的符号，例如弯引号和 €（欧元）；有些 Windows 应用把它称为“ANSI”，但它并不是 ANSI 标准。

`cp437`

　　IBM PC 最初的字符集，包含框图符号。与后来出现的 `latin1` 不兼容。

`gb2312`

　　用于编码简体中文的陈旧标准；这是亚洲语言中使用较广泛的多字节编码之一。

`utf-8`

　　目前 Web 中最常见的 8 位编码；[3] 与 ASCII 兼容（纯 ASCII 文本是有效的 UTF-8 文本）。

> [3]W3Techs 发布的“Usage of character encodings for websites”（https://w3techs.com/technologies/overview/character_encoding/all）报告指出，截至 2014 年 9 月，81.4% 的网站使用 UTF-8；而 Built With 发布的“Encoding Usage Statistics”（http://trends.builtwith.com/encoding）估计的比例则是 79.4%。

`utf-16le`

UTF-16 的 16 位编码方案的一种形式；所有 UTF-16 支持通过转义序列（称为“代理对”，surrogate pair）表示超过 U+FFFF 的码位。

UTF-16 取代了 1996 年发布的 Unicode 1.0 编码（UCS-2）。这个编码在很多系统中仍在使用，但是支持的最大码位是 U+FFFF。从 Unicode 6.3 起，分配的码位中有超过 50% 在 U+10000 以上，包括逐渐流行的表情符号（emoji pictograph）。

概述常规的编码之后，下面要处理编码和解码过程中存在的问题。

4.4　了解编解码问题

虽然有个一般性的 UnicodeError 异常，但是报告错误时几乎都会指明具体的异常：UnicodeEncodeError（把字符串转换成二进制序列时）或 UnicodeDecodeError（把二进制序列转换成字符串时）。如果源码的编码与预期不符，加载 Python 模块时还可能抛出 SyntaxError。接下来的几节说明如何处理这些错误。

> 出现与 Unicode 有关的错误时，首先要明确异常的类型。导致编码问题的是 UnicodeEncodeError、UnicodeDecodeError，还是如 SyntaxError 的其他错误？解决问题之前必须清楚这一点。

4.4.1　处理UnicodeEncodeError

多数非 UTF 编解码器只能处理 Unicode 字符的一小部分子集。把文本转换成字节序列时，如果目标编码中没有定义某个字符，那就会抛出 UnicodeEncodeError 异常，除非把 errors 参数传给编码方法或函数，对错误进行特殊处理。处理错误的方式如示例 4-6 所示。

示例 **4-6**　编码成字节序列：成功和错误处理

```
>>> city = 'São Paulo'
>>> city.encode('utf_8') ❶
b'S\xc3\xa3o Paulo'
>>> city.encode('utf_16')
b'\xff\xfeS\x00\xe3\x00o\x00 \x00P\x00a\x00u\x00l\x00o\x00'
>>> city.encode('iso8859_1') ❷
b'S\xe3o Paulo'
>>> city.encode('cp437') ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/.../lib/python3.4/encodings/cp437.py", line 12, in encode
    return codecs.charmap_encode(input,errors,encoding_map)
UnicodeEncodeError: 'charmap' codec can't encode character '\xe3' in
position 1: character maps to <undefined>
>>> city.encode('cp437', errors='ignore') ❹
```

```
b'So Paulo'
>>> city.encode('cp437', errors='replace') ❺
b'S?o Paulo'
>>> city.encode('cp437', errors='xmlcharrefreplace') ❻
b'S&#227;o Paulo'
```

❶ 'utf_?' 编码能处理任何字符串。

❷ 'iso8859_1' 编码也能处理字符串 'São Paulo'。

❸ 'cp437' 无法编码 'ã'（带波形符的“a”）。默认的错误处理方式 'strict' 抛出 UnicodeEncodeError。

❹ error='ignore' 处理方式悄无声息地跳过无法编码的字符；这样做通常很是不妥。

❺ 编码时指定 error='replace'，把无法编码的字符替换成 '?'；数据损坏了，但是用户知道出了问题。

❻ 'xmlcharrefreplace' 把无法编码的字符替换成 XML 实体。

> 编解码器的错误处理方式是可扩展的。你可以为 errors 参数注册额外的字符串，方法是把一个名称和一个错误处理函数传给 codecs.register_error 函数。参见 codecs.register_error 函数的文档（https://docs.python.org/3/library/codecs.html#codecs.register_error）。

4.4.2　处理UnicodeDecodeError

不是每一个字节都包含有效的 ASCII 字符，也不是每一个字符序列都是有效的 UTF-8 或 UTF-16。因此，把二进制序列转换成文本时，如果假设是这两个编码中的一个，遇到无法转换的字节序列时会抛出 UnicodeDecodeError。

另一方面，很多陈旧的 8 位编码——如 'cp1252'、'iso8859_1' 和 'koi8_r'——能解码任何字节序列流而不抛出错误，例如随机噪声。因此，如果程序使用错误的 8 位编码，解码过程悄无声息，而得到的是

无用输出。

乱码字符称为鬼符（gremlin）或 mojibake（文字化け，“变形文本”的日文）。

示例 4-7 演示了使用错误的编解码器可能出现鬼符或抛出 `UnicodeDecodeError`。

示例 **4-7**　把字节序列解码成字符串：成功和错误处理

```
>>> octets = b'Montr\xe9al'  ❶
>>> octets.decode('cp1252')  ❷
'Montréal'
>>> octets.decode('iso8859_7')  ❸
'Montrιal'
>>> octets.decode('koi8_r')  ❹
'MontrИal'
>>> octets.decode('utf_8')  ❺
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe9 in position 5:
invalid continuation byte
>>> octets.decode('utf_8', errors='replace')  ❻
'Montral'
```

❶ 这些字节序列是使用 `latin1` 编码的“Montréal”；`'\xe9'` 字节对应“é”。

❷ 可以使用 `'cp1252'`（Windows 1252）解码，因为它是 `latin1` 的有效超集。

❸ ISO-8859-7 用于编码希腊文，因此无法正确解释 `'\xe9'` 字节，而且没有抛出错误。

❹ KOI8-R 用于编码俄文；这里，`'\xe9'` 表示西里尔字母“И”。

❺ `'utf_8'` 编解码器检测到 `octets` 不是有效的 UTF-8 字符串，抛出 `UnicodeDecodeError`。

❻ 使用 `'replace'` 错误处理方式，`\xe9` 替换成了“�”（码位是

U+FFFD），这是官方指定的 REPLACEMENT CHARACTER（替换字符），表示未知字符。

4.4.3　使用预期之外的编码加载模块时抛出的 SyntaxError

Python 3 默认使用 UTF-8 编码源码，Python 2（从 2.5 开始）则默认使用 ASCII。如果加载的 .py 模块中包含 UTF-8 之外的数据，而且没有声明编码，会得到类似下面的消息：

```
SyntaxError: Non-UTF-8 code starting with '\xe1' in file ola.py on line
  1, but no encoding declared; see http://python.org/dev/peps/pep-0263/
  for details
```

GNU/Linux 和 OS X 系统大都使用 UTF-8，因此打开在 Windows 系统中使用 cp1252 编码的 .py 文件时可能发生这种情况。注意，这个错误在 Windows 版 Python 中也可能会发生，因为 Python 3 为所有平台设置的默认编码都是 UTF-8。

为了修正这个问题，可以在文件顶部添加一个神奇的 coding 注释，如示例 4-8 所示。

示例 **4-8**　ola.py：“你好，世界！”的葡萄牙语版

```
# coding: cp1252

print('Olá, Mundo!')
```

现在，Python 3 的源码不再限于使用 ASCII，而是默认使用优秀的 UTF-8 编码，因此要修正源码的陈旧编码（如 'cp1252'）问题，最好将其转换成 UTF-8，别去麻烦 coding 注释。如果你用的编辑器不支持 UTF-8，那么是时候换一个了。

源码中能不能使用非 **ASCII** 名称

Python 3 允许在源码中使用非 ASCII 标识符：

```
>>> ação = 'PBR'  # ação = stock
>>> ε = 10**-6    # ε = epsilon
```

有些人不喜欢这么做。支持始终使用 ASCII 标识符的人认为，这样便于所有人阅读和编辑代码。这些人没切中要害：源码应该便于目标群体阅读和编辑，而不是“所有人”。如果代码属于跨国公司，或者是开源的，想让来自世界各地的人作贡献，那么标识符应该使用英语，也就是说只能使用 ASCII 字符。

但是，如果你是巴西的一位老师，那么使用葡萄牙语正确拼写变量和函数名更便于学生阅读代码。而且，这些学生在本地化的键盘中不难打出变音符号和重音元音字母。

现在，Python 能解析 Unicode 名称，而且源码的默认编码是 UTF-8，我觉得没有任何理由使用不带重音符号的葡萄牙语编写标识符。在 Python 2 中确实不能这么做，除非你也想使用 Python 2 运行代码，否则不必如此。如果使用葡萄牙语命名标识符却不带重音符号的话，这样写出的代码对任何人来说都不易阅读。

这是我作为说葡萄牙语的巴西人的观点，不过我相信也适用于其他国家和文化：选择对团队而言易于阅读的人类语言，然后使用正确的字符拼写。

假如有个文本文件，里面保存的是源码或诗句，但是你不知道它的编码。如何查明真正的编码呢？下一节使用一个推荐的库回答这个问题。

4.4.4 如何找出字节序列的编码

如何找出字节序列的编码？简单来说，不能。必须有人告诉你。

有些通信协议和文件格式，如 HTTP 和 XML，包含明确指明内容编码的首部。可以肯定的是，某些字节流不是 ASCII，因为其中包含大于 127 的字节值，而且制定 UTF-8 和 UTF-16 的方式也限制了可用的字节序列。不过即便如此，我们也不能根据特定的位模式来 100% 确定二进制文件的编码是 ASCII 或 UTF-8。

然而，就像人类语言也有规则和限制一样，只要假定字节流是人类可读的纯文本，就可能通过试探和分析找出编码。例如，如果 `b'\x00'` 字节经常出现，那么可能是 16 位或 32 位编码，而不是 8 位编码方案，因为纯文本中不能包含空字符；如果字节序列 `b'\x20\x00'` 经常出现，那么可能是 UTF-16LE 编码中的空格字符（U+0020），而不是鲜为人知的 U+2000 `EN QUAD` 字符——谁知道这是什么呢！

统一字符编码侦测包 Chardet（https://pypi.python.org/pypi/chardet）就是这样工作的，它能识别所支持的 30 种编码。Chardet 是一个 Python 库，可以在程序中使用，不过它也提供了命令行工具 `chardetect`。下面是它对本章书稿文件的检测报告：

```
$ chardetect 04-text-byte.asciidoc
04-text-byte.asciidoc: utf-8 with confidence 0.99
```

二进制序列编码文本通常不会明确指明自己的编码，但是 UTF 格式可以在文本内容的开头添加一个字节序标记。参见下一节。

4.4.5　BOM：有用的鬼符

在示例 4-5 中，你可能注意到了，UTF-16 编码的序列开头有几个额外的字节，如下所示：

```
>>> u16 = 'El Niño'.encode('utf_16')
>>> u16
b'\xff\xfeE\x00l\x00 \x00N\x00i\x00\xf1\x00o\x00'
```

我指的是 `b'\xff\xfe'`。这是 BOM，即字节序标记（byte-order mark），指明编码时使用 Intel CPU 的小字节序。

在小字节序设备中，各个码位的最低有效字节在前面：字母 `'E'` 的码位是 U+0045（十进制数 69），在字节偏移的第 2 位和第 3 位编码为 69 和 0。

```
>>> list(u16)
[255, 254, 69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111, 0]
```

在大字节序 CPU 中，编码顺序是相反的；'E' 编码为 0 和 69。

为了避免混淆，UTF-16 编码在要编码的文本前面加上特殊的不可见字符 ZERO WIDTH NO-BREAK SPACE（U+FEFF）。在小字节序系统中，这个字符编码为 b'\xff\xfe'（十进制数 255, 254）。因为按照设计，U+FFFE 字符不存在，在小字节序编码中，字节序列 b'\xff\xfe' 必定是 ZERO WIDTH NO-BREAK SPACE，所以编解码器知道该用哪个字节序。

UTF-16 有两个变种：UTF-16LE，显式指明使用小字节序；UTF-16BE，显式指明使用大字节序。如果使用这两个变种，不会生成 BOM：

```
>>> u16le = 'El Niño'.encode('utf_16le')
>>> list(u16le)
[69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111, 0]
>>> u16be = 'El Niño'.encode('utf_16be')
>>> list(u16be)
[0, 69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111]
```

如果有 BOM，UTF-16 编解码器会将其过滤掉，为你提供没有前导 ZERO WIDTH NO-BREAK SPACE 字符的真正文本。根据标准，如果文件使用 UTF-16 编码，而且没有 BOM，那么应该假定它使用的是 UTF-16BE（大字节序）编码。然而，Intel x86 架构用的是小字节序，因此有很多文件用的是不带 BOM 的小字节序 UTF-16 编码。

与字节序有关的问题只对一个字（word）占多个字节的编码（如 UTF-16 和 UTF-32）有影响。UTF-8 的一大优势是，不管设备使用哪种字节序，生成的字节序列始终一致，因此不需要 BOM。尽管如此，某些 Windows 应用（尤其是 Notepad）依然会在 UTF-8 编码的文件中添加 BOM；而且，Excel 会根据有没有 BOM 确定文件是不是 UTF-8 编码，否则，它假设内容使用 Windows 代码页（codepage）编码。UTF-8 编码的 U+FEFF 字符是一个三字节序列：b'\xef\xbb\xbf'。因此，如果文件以这三个字节开头，有可能是带有 BOM 的 UTF-8 文件。然而，Python 不会因为文件以 b'\xef\xbb\xbf' 开头就自动假定它是 UTF-8 编码的。

下面换个话题，讨论 Python 3 处理文本文件的方式。

4.5 处理文本文件

处理文本的最佳实践是“Unicode 三明治”（如图 4-2 所示）。[4] 意思是，要尽早把输入（例如读取文件时）的字节序列解码成字符串。这种三明治中的“肉片”是程序的业务逻辑，在这里只能处理字符串对象。在其他处理过程中，一定不能编码或解码。对输出来说，则要尽量晚地把字符串编码成字节序列。多数 Web 框架都是这样做的，使用框架时很少接触字节序列。例如，在 Django 中，视图应该输出 Unicode 字符串；Django 会负责把响应编码成字节序列，而且默认使用 UTF-8 编码。

> [4]我第一次见到“Unicode 三明治”这种说法是在 Ned Batchelder 在 US PyCon 2012 上所做的精彩演讲中：“Pragmatic Unicode”（http://nedbatchelder.com/text/unipain/unipain.html）。



图 **4-2**：**Unicode** 三明治——目前处理文本的最佳实践

在 Python 3 中能轻松地采纳 Unicode 三明治的建议，因为内置的 `open` 函数会在读取文件时做必要的解码，以文本模式写入文件时还会做必要的编码，所以调用 `my_file.read()` 方法得到的以及传给 `my_file.write(text)` 方法的都是字符串对象。[5]

> [5]Python 2.6 或 Python 2.7 用户要使用 `io.open()` 函数才能得到读写文件时自动执行的解码和编码操作。

可以看出，处理文本文件很简单。但是，如果依赖默认编码，你会遇到麻烦。

看一下示例 4-9 中的控制台会话。你能发现问题吗？

示例 **4-9**　一个平台上的编码问题（如果在你的机器上运行，它可能会发生，也可能不会）

```
>>> open('cafe.txt', 'w', encoding='utf_8').write('café')
4
>>> open('cafe.txt').read()
'cafÃ©'
```

问题是：写入文件时指定了 UTF-8 编码，但是读取文件时没有这么做，因此 Python 假定要使用系统默认的编码（Windows 1252），于是文件的最后一个字节解码成了字符 `'Ã©'`，而不是 `'é'`。

我是在 Windows 7 中运行示例 4-9 的。在新版 GNU/Linux 或 Mac OS X 中运行同样的语句不会出问题，因为这几个操作系统的默认编码是 UTF-8，让人误以为一切正常。如果打开文件是为了写入，但是没有指定编码参数，会使用区域设置中的默认编码，而且使用那个编码也能正确读取文件。但是，如果脚本要生成文件，而字节的内容取决于平台或同一平台中的区域设置，那么就可能导致兼容问题。

需要在多台设备中或多种场合下运行的代码，一定不能依赖默认编码。打开文件时始终应该明确传入 `encoding=` 参数，因为不同的设备使用的默认编码可能不同，有时隔一天也会发生变化。

示例 4-9 中有个奇怪的细节：第一个语句中的 `write` 函数报告写入了 4 个字符，但是下一行读取时却得到了 5 个字符。示例 4-10 是对示例 4-9 的扩展，对这个问题以及其他细节做了说明。

示例 **4-10**　仔细分析在 Windows 中运行的示例 4-9，找出并修正问题

```
>>> fp = open('cafe.txt', 'w', encoding='utf_8')
>>> fp  ❶
<_io.TextIOWrapper name='cafe.txt' mode='w' encoding='utf_8'>
```

```
>>> fp.write('café')
4  ❷
>>> fp.close()
>>> import os
>>> os.stat('cafe.txt').st_size
5  ❸
>>> fp2 = open('cafe.txt')
>>> fp2  ❹
<_io.TextIOWrapper name='cafe.txt' mode='r' encoding='cp1252'>
>>> fp2.encoding  ❺
'cp1252'
>>> fp2.read()
'cafÃ©'  ❻
>>> fp3 = open('cafe.txt', encoding='utf_8')  ❼
>>> fp3
<_io.TextIOWrapper name='cafe.txt' mode='r' encoding='utf_8'>
>>> fp3.read()
'café'  ❽
>>> fp4 = open('cafe.txt', 'rb')  ❾
>>> fp4
<_io.BufferedReader name='cafe.txt'>  ❿
>>> fp4.read()  ⓫
b'caf\xc3\xa9'
```

❶ 默认情况下，`open` 函数采用文本模式，返回一个 `TextIOWrapper` 对象。

❷ 在 `TextIOWrapper` 对象上调用 `write` 方法返回写入的 Unicode 字符数。

❸ `os.stat` 报告文件中有 5 个字节；UTF-8 编码的 `'é'` 占两个字节，0xc3 和 0xa9。

❹ 打开文本文件时没有显式指定编码，返回一个 `TextIOWrapper` 对象，编码是区域设置中的默认值。

❺ `TextIOWrapper` 对象有个 `encoding` 属性；查看它，发现这里的编码是 `cp1252`。

❻ 在 Windows `cp1252` 编码中，0xc3 字节是“Ã”（带波形符的 A），0xa9 字节是版权符号。

❼ 使用正确的编码打开那个文件。

❽ 结果符合预期：得到的是四个 Unicode 字符 `'café'`。

❾ `'rb'` 标志指明在二进制模式中读取文件。

❿ 返回的是 `BufferedReader` 对象，而不是 `TextIOWrapper` 对象。

⓫读取返回的字节序列，结果与预期相符。

> 除非想判断编码，否则不要在二进制模式中打开文本文件；即便如此，也应该使用 Chardet，而不是重新发明轮子（参见 4.4.4 节）。常规代码只应该使用二进制模式打开二进制文件，如光栅图像。

示例 4-10 的问题是，打开文本文件时依赖默认设置。默认设置有许多来源，参见下一节。

编码默认值：一团糟

有几个设置对 Python I/O 的编码默认值有影响，如示例 4-11 中的 default_encodings.py 脚本所示。

示例 **4-11**　探索编码默认值

```
import sys, locale

expressions = """
        locale.getpreferredencoding()
        type(my_file)
        my_file.encoding
        sys.stdout.isatty()
        sys.stdout.encoding
        sys.stdin.isatty()
        sys.stdin.encoding
        sys.stderr.isatty()
        sys.stderr.encoding
        sys.getdefaultencoding()
        sys.getfilesystemencoding()
    """
```

```
my_file = open('dummy', 'w')

for expression in expressions.split():
    value = eval(expression)
    print(expression.rjust(30), '->', repr(value))
```

示例 4-11 在 GNU/Linux（Ubuntu 14.04）和 OS X（Mavericks 10.9）中的输出一样，表明这些系统中始终使用 `UTF-8`：

```
$ python3 default_encodings.py
 locale.getpreferredencoding() -> 'UTF-8'
                type(my_file) -> <class '_io.TextIOWrapper'>
             my_file.encoding -> 'UTF-8'
          sys.stdout.isatty() -> True
          sys.stdout.encoding -> 'UTF-8'
           sys.stdin.isatty() -> True
           sys.stdin.encoding -> 'UTF-8'
          sys.stderr.isatty() -> True
          sys.stderr.encoding -> 'UTF-8'
    sys.getdefaultencoding() -> 'utf-8'
 sys.getfilesystemencoding() -> 'utf-8'
```

然而，在 Windows 中的输出有所不同，如示例 4-12 所示。

示例 **4-12** 在Windows 7（SP1）巴西版中的 cmd.exe 中输出的默认编码；PowerShell 输出的结果相同

```
Z:\>chcp  ❶
Página de código ativa: 850
Z:\>python default_encodings.py  ❷
 locale.getpreferredencoding() -> 'cp1252'  ❸
                type(my_file) -> <class '_io.TextIOWrapper'>
             my_file.encoding -> 'cp1252'  ❹
          sys.stdout.isatty() -> True      ❺
          sys.stdout.encoding -> 'cp850'   ❻
           sys.stdin.isatty() -> True
           sys.stdin.encoding -> 'cp850'
          sys.stderr.isatty() -> True
          sys.stderr.encoding -> 'cp850'
    sys.getdefaultencoding() -> 'utf-8'
 sys.getfilesystemencoding() -> 'mbcs'
```

❶ chcp 输出当前控制台激活的代码页：850。

❷ 运行 default_encodings.py，把结果输出到控制台。

❸ locale.getpreferredencoding() 是最重要的设置。

❹ 文本文件默认使用 locale.getpreferredencoding()。

❺ 输出到控制台中，因此 sys.stdout.isatty() 返回 True。

❻ 因此，sys.stdout.encoding 与控制台的编码相同。

如果把输出重定向到文件，如下所示：

```
Z:\>python default_encodings.py > encodings.log
```

sys.stdout.isatty() 的返回值会变成 False，sys.stdout.encoding 会设为 locale.getpreferredencoding()，在那台设备中是 'cp1252'。

注意，示例 4-12 中有 4 种不同的编码。

- 如果打开文件时没有指定 encoding 参数，默认值由 locale.getpreferredencoding() 提供（在示例 4-12 中是 'cp1252'）。

- 如果设定了 PYTHONIOENCODING 环境变量（https://docs.python.org/3/using/cmdline.html#envvar-PYTHONIOENCODING），sys.stdout/stdin/stderr 的编码使用设定的值；否则，继承自所在的控制台；如果输入 / 输出重定向到文件，则由 locale.getpreferredencoding() 定义。

- Python 在二进制数据和字符串之间转换时，内部使用 sys.getdefaultencoding() 获得的编码；Python 3 很少如此，但仍有发生。[6] 这个设置不能修改。[7]

- `sys.getfilesystemencoding()` 用于编解码文件名（不是文件内容）。把字符串参数作为文件名传给 `open()` 函数时就会使用它；如果传入的文件名参数是字节序列，那就不经改动直接传给 OS API。“Unicode HOWTO”一文（https://docs.python.org/3/howto/unicode.html）中说：“在 Windows 中，Python 使用 `mbcs` 这个名称引用当前配置的编码。”MBCS 是 Multi Byte Character Set（多字节字符集）的首字母缩写，在 Windows 中是陈旧的变长编码，如 `gb2312` 或 `Shift_JIS`，而不是 UTF-8。[关于这个话题，Stack Overflow 中有一个很好的回答“，Difference between MBCS and UTF-8 on Windows”（http://stackoverflow.com/questions/3298569/difference-between-mbcs-and-utf-8-on-windows）。]

[6]研究这个话题时，我在 Python 内部找不到把字节序列转换成字符串的情况。Python 核心开发者 Antoine Pitrou 在 `comp.python.devel` 邮件列表中说（http://article.gmane.org/gmane.comp.python.devel/110036），CPython 的内部函数“在 py3k 中很少这么做”。

[7]Python 2 对 `sys.setdefaultencoding` 函数的使用方式不当，Python 3 的文档中已经没有这个函数。这个函数是供核心开发者使用的，用于在内部的默认编码未定时设置编码。在 `comp.python.devel` 邮件列表的那个话题中（http://article.gmane.org/gmane.comp.python.devel/109916），Marc-André Lemburg 说，用户代码一定不能调用 `sys.setdefaultencoding` 函数，而且对 CPython 来说，它的值在 Python 2 中只能是 `'ascii'`，在 Python 3 中只能是 `'utf-8'`。

在 GNU/Linux 和 OS X 中，这些编码的默认值都是 UTF-8，而且多年来都是如此，因此 I/O 能处理所有 Unicode 字符。在 Windows 中，不仅同一个系统中使用不同的编码，还有只支持 ASCII 和 127 个额外的字符的代码页（如 `'cp850'` 或 `'cp1252'`），而且不同的代码页之间增加的字符也有所不同。因此，若不多加小心，Windows 用户更容易遇到编码问题。

综上，`locale.getpreferredencoding()` 返回的编码是最重要的：这是打开文件的默认编码，也是重定向到文件的 `sys.stdout/stdin/stderr` 的默认编码。然而，文档也说道（摘录部分，https://docs.python.org/3/library/locale.html#locale.getpreferredencoding

```
locale.getpreferredencoding(do_setlocale=True)
```

> 根据用户的偏好设置，返回文本数据的编码。用户的偏好设置在不同系统中的设定方式不同，而且在某些系统中可能无法通过编程方式设置，因此这个函数返回的只是猜测的编码……

因此，关于编码默认值的最佳建议是：别依赖默认值。

如果遵从 Unicode 三明治的建议，而且始终在程序中显式指定编码，那将避免很多问题。可惜，即使把字节序列正确地转换成字符串，Unicode 仍有不尽如人意的地方。接下来的两节讨论的话题对 ASCII 世界来说很简单，但是在 Unicode 领域就变得相当复杂：文本规范化（即为了比较而把文本转换成统一的表述）和排序。

4.6　为了正确比较而规范化Unicode字符串

因为 Unicode 有组合字符（变音符号和附加到前一个字符上的记号，打印时作为一个整体），所以字符串比较起来很复杂。

例如，“café”这个词可以使用两种方式构成，分别有 4 个和 5 个码位，但是结果完全一样：

```
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> s1, s2
('café', 'café')
>>> len(s1), len(s2)
(4, 5)
>>> s1 == s2
False
```

U+0301 是 `COMBINING ACUTE ACCENT`，加在“e”后面得到“é”。在 Unicode 标准中，`'é'` 和 `'e\u0301'` 这样的序列叫“标准等价物”（canonical equivalent），应用程序应该把它们视作相同的字符。但是，Python 看到的是不同的码位序列，因此判定二者不相等。

这个问题的解决方案是使用 `unicodedata.normalize` 函数提供的 Unicode 规范化。这个函数的第一个参数是这 4 个字符串中的一个：`'NFC'`、`'NFD'`、`'NFKC'` 和 `'NFKD'`。下面先说明前两个。

NFC（Normalization Form C）使用最少的码位构成等价的字符串，而 NFD 把组合字符分解成基字符和单独的组合字符。这两种规范化方式都能让比较行为符合预期：

```
>>> from unicodedata import normalize
>>> s1 = 'café'  # 把"e"和重音符组合在一起
>>> s2 = 'cafe\u0301'  # 分解成"e"和重音符
>>> len(s1), len(s2)
(4, 5)
>>> len(normalize('NFC', s1)), len(normalize('NFC', s2))
```

```
(4, 4)
>>> len(normalize('NFD', s1)), len(normalize('NFD', s2))
(5, 5)
>>> normalize('NFC', s1) == normalize('NFC', s2)
True
>>> normalize('NFD', s1) == normalize('NFD', s2)
True
```

西方键盘通常能输出组合字符，因此用户输入的文本默认是 NFC 形式。不过，安全起见，保存文本之前，最好使用 `normalize('NFC', user_text)` 清洗字符串。NFC 也是 W3C 的"Character Model for the World Wide Web: String Matching and Searching"规范（https://www.w3.org/TR/charmod-norm/）推荐的规范化形式。

使用 NFC 时，有些单字符会被规范成另一个单字符。例如，电阻的单位欧姆（Ω）会被规范成希腊字母大写的欧米加。这两个字符在视觉上是一样的，但是比较时并不相等，因此要规范化，防止出现意外：

```
>>> from unicodedata import normalize, name
>>> ohm = '\u2126'
>>> name(ohm)
'OHM SIGN'
>>> ohm_c = normalize('NFC', ohm)
>>> name(ohm_c)
'GREEK CAPITAL LETTER OMEGA'
>>> ohm == ohm_c
False
>>> normalize('NFC', ohm) == normalize('NFC', ohm_c)
True
```

在另外两个规范化形式（NFKC 和 NFKD）的首字母缩略词中，字母 K 表示"compatibility"（兼容性）。这两种是较严格的规范化形式，对"兼容字符"有影响。虽然 Unicode 的目标是为各个字符提供"规范的"码位，但是为了兼容现有的标准，有些字符会出现多次。例如，虽然希腊字母表中有"μ"这个字母（码位是 U+03BC，`GREEK SMALL LETTER MU`），但是 Unicode 还是加入了微符号 `'µ'`（U+00B5），以便与 `latin1` 相互转换。因此，微符号是一个"兼容字符"。

在 NFKC 和 NFKD 形式中，各个兼容字符会被替换成一个或多个"兼容分解"字符，即便这样有些格式损失，但仍是"首选"表述——理想情况下，格式化是外部标记的职责，不应该由 Unicode 处理。下面举个例

子。二分之一 '½'（U+00BD）经过兼容分解后得到的是三个字符序列 '1/2'；微符号 'µ'（U+00B5）经过兼容分解后得到的是小写字母 'μ'（U+03BC）。[8]

[8]微符号是"兼容字符"，而欧姆符号不是，这还真是奇怪。因此，NFC 不会改动微符号，但是会把欧姆符号改成大写的欧米加；而 NFKC 和 NFKD 会把欧姆和微符号都改成其他字符。

下面是 NFKC 的具体应用：

```
>>> from unicodedata import normalize, name
>>> half = '½'
>>> normalize('NFKC', half)
'1/2'
>>> four_squared = '4²'
>>> normalize('NFKC', four_squared)
'42'
>>> micro = 'µ'
>>> micro_kc = normalize('NFKC', micro)
>>> micro, micro_kc
('µ', 'μ')
>>> ord(micro), ord(micro_kc)
(181, 956)
>>> name(micro), name(micro_kc)
('MICRO SIGN', 'GREEK SMALL LETTER MU')
```

使用 '1/2' 替代 '½' 可以接受，微符号也确实是小写的希腊字母 'μ'，但是把 '4²' 转换成 '42' 就改变原意了。某些应用程序可以把 '4²' 保存为 '4²'，但是 normalize 函数对格式一无所知。因此，NFKC 或 NFKD 可能会损失或曲解信息，但是可以为搜索和索引提供便利的中间表述：用户搜索 '1 / 2 inch' 时，如果还能找到包含 '½ inch' 的文档，那么用户会感到满意。

使用 NFKC 和 NFKD 规范化形式时要小心，而且只能在特殊情况中使用，例如搜索和索引，而不能用于持久存储，因为这两种转换会导致数据损失。

为搜索或索引准备文本时，还有一个有用的操作，即下一节讨论的大小写折叠。

4.6.1 大小写折叠

大小写折叠其实就是把所有文本变成小写，再做些其他转换。这个功能由 `str.casefold()` 方法（Python 3.3 新增）支持。

对于只包含 `latin1` 字符的字符串 `s`，`s.casefold()` 得到的结果与 `s.lower()` 一样，唯有两个例外：微符号 `'µ'` 会变成小写的希腊字母“μ”（在多数字体中二者看起来一样）；德语 Eszett（“sharp s”，ß）会变成“ss”。

```
>>> micro = 'µ'
>>> name(micro)
'MICRO SIGN'
>>> micro_cf = micro.casefold()
>>> name(micro_cf)
'GREEK SMALL LETTER MU'
>>> micro, micro_cf
('µ', 'μ')
>>> eszett = 'ß'
>>> name(eszett)
'LATIN SMALL LETTER SHARP S'
>>> eszett_cf = eszett.casefold()
>>> eszett, eszett_cf
('ß', 'ss')
```

自 Python 3.4 起，`str.casefold()` 和 `str.lower()` 得到不同结果的有 116 个码位。Unicode 6.3 命名了 110 122 个字符，这只占 0.11%。

与 Unicode 相关的任何问题一样，大小写折叠是个复杂的问题，有很多语言上的特殊情况，但是 Python 核心团队尽力提供了一种方案，能满足大多数用户的需求。

接下来的几节将使用这些规范化知识来开发几个实用的函数。

4.6.2 规范化文本匹配实用函数

由前文可知，NFC 和 NFD 可以放心使用，而且能合理比较 Unicode 字符串。对大多数应用来说，NFC 是最好的规范化形式。不区分大小写的比较应该使用 `str.casefold()`。

如果要处理多语言文本，工具箱中应该有示例 4-13 中的 `nfc_equal` 和 `fold_equal` 函数。

示例 **4-13**　normeq.py：比较规范化 Unicode 字符串

```
"""
Utility functions for normalized Unicode string comparison.

Using Normal Form C, case sensitive:

    >>> s1 = 'café'
    >>> s2 = 'cafe\u0301'
    >>> s1 == s2
    False
    >>> nfc_equal(s1, s2)
    True
    >>> nfc_equal('A', 'a')
    False

Using Normal Form C with case folding:

    >>> s3 = 'Straße'
    >>> s4 = 'strasse'
    >>> s3 == s4
    False
    >>> nfc_equal(s3, s4)
    False
    >>> fold_equal(s3, s4)
    True
    >>> fold_equal(s1, s2)
    True
    >>> fold_equal('A', 'a')
    True

"""

from unicodedata import normalize

def nfc_equal(str1, str2):
    return normalize('NFC', str1) == normalize('NFC', str2)

def fold_equal(str1, str2):
    return (normalize('NFC', str1).casefold() ==
            normalize('NFC', str2).casefold())
```

除了 Unicode 规范化和大小写折叠（二者都是 Unicode 标准的一部分）之外，有时需要进行更为深入的转换，例如把 `'café'` 变成 `'cafe'`。下一节说明何时以及如何进行这种转换。

4.6.3　极端“规范化”：去掉变音符号

Google 搜索涉及很多技术，其中一个显然是忽略变音符号（如重音符、下加符等），至少在某些情况下会这么做。去掉变音符号不是正确的规范化方式，因为这往往会改变词的意思，而且可能误判搜索结果。但是对现实生活却有所帮助：人们有时很懒，或者不知道怎么正确使用变音符号，而且拼写规则会随时间变化，因此实际语言中的重音经常变来变去。

除了搜索，去掉变音符号还能让 URL 更易于阅读，至少对拉丁语系语言是如此。下面是维基百科中介绍圣保罗市（São Paulo）的文章的 URL：

```
http://en.wikipedia.org/wiki/S%C3%A3o_Paulo
```

其中，“%C3%A3”是 UTF-8 编码“ã”字母（带有波形符的“a”）转义后得到的结果。下述形式更友好，尽管拼写是错误的：

```
http://en.wikipedia.org/wiki/Sao_Paulo
```

如果想把字符串中的所有变音符号都去掉，可以使用示例 4-14 中的函数。

示例 **4-14**　去掉全部组合记号的函数（在 sanitize.py 模块中）

```
import unicodedata
import string


def shave_marks(txt):
    """去掉全部变音符号"""
    norm_txt = unicodedata.normalize('NFD', txt)  ❶
    shaved = ''.join(c for c in norm_txt
```

```
                 if not unicodedata.combining(c))  ❷
    return unicodedata.normalize('NFC', shaved)  ❸
```

❶ 把所有字符分解成基字符和组合记号。

❷ 过滤掉所有组合记号。

❸ 重组所有字符。

示例 4-15 是 `shave_marks` 函数的两个使用示例。

示例 **4-15**　示例 4-14 中 `shave_marks` 函数的两个使用示例

```
>>> order = '“Herr Voß: • ½ cup of OEtker™ caffè latte • bowl of açaí.”'
>>> shave_marks(order)
'“Herr Voß: • ½ cup of OEtker™ caffe latte • bowl of acai.”'  ❶
>>> Greek = 'Ζέφυρος, Zéfiro'
>>> shave_marks(Greek)
'Ζεφυρος, Zefiro'  ❷
```

❶ 只替换了“è”“ç”和“í”三个字符。

❷ “έ”和“é”都被替换了。

示例 4-14 中定义的 `shave_marks` 函数使用起来没问题，但是也许做得太多了。通常，去掉变音符号是为了把拉丁文本变成纯粹的 ASCII，但是 `shave_marks` 函数还会修改非拉丁字符（如希腊字母），而只去掉重音符并不能把它们变成 ASCII 字符。因此，我们应该分析各个基字符，仅当字符在拉丁字母表中时才删除附加的记号，如示例 4-16 所示。

示例 **4-16**　删除拉丁字母中组合记号的函数（`import` 语句省略了，因为这是示例 4-14 中定义的 sanitize.py 模块的一部分）

```
def shave_marks_latin(txt):
    """把拉丁基字符中所有的变音符号删除"""
    norm_txt = unicodedata.normalize('NFD', txt)  ❶
    latin_base = False
    keepers = []
    for c in norm_txt:
```

```
        if unicodedata.combining(c) and latin_base:   ❷
            continue  # 忽略拉丁基字符上的变音符号
        keepers.append(c)                             ❸
        # 如果不是组合字符，那就是新的基字符
        if not unicodedata.combining(c):              ❹
            latin_base = c in string.ascii_letters
    shaved = ''.join(keepers)
    return unicodedata.normalize('NFC', shaved)       ❺
```

❶ 把所有字符分解成基字符和组合记号。

❷ 基字符为拉丁字母时，跳过组合记号。

❸ 否则，保存当前字符。

❹ 检测新的基字符，判断是不是拉丁字母。

❺ 重组所有字符。

更彻底的规范化步骤是把西文文本中的常见符号（如弯引号、长破折号、项目符号，等等）替换成 ASCII 中的对等字符。示例 4-17 中的 `asciize` 函数就是这么做的。

示例 **4-17**　把一些西文印刷字符转换成 ASCII 字符（这个代码片段也是示例 4-14 中 sanitize.py 模块的一部分）

```
single_map = str.maketrans("""‚ƒ„†ˆ‹‘’“”•–—˜›""",  ❶
                           """'f"*^<''""---~>""")

multi_map = str.maketrans({ ❷
    '€': '<euro>',
    '…': '...',
    'OE': 'OE',
    '™': '(TM)',
    'oe': 'oe',
    '‰': '<per mille>',
    '‡': '**',
})

multi_map.update(single_map)  ❸
```

```
def dewinize(txt):
    """把Win1252符号替换成ASCII字符或序列"""
    return txt.translate(multi_map)  ❹

def asciize(txt):
    no_marks = shave_marks_latin(dewinize(txt))     ❺
    no_marks = no_marks.replace('ß', 'ss')          ❻
    return unicodedata.normalize('NFKC', no_marks)  ❼
```

❶ 构建字符替换字符的映射表。

❷ 构建字符替换字符串的映射表。

❸ 合并两个映射表。

❹ `dewinize` 函数不影响 `ASCII` 或 `latin1` 文本，只替换 Microsoft 在 `cp1252` 中为 `latin1` 额外添加的字符。

❺ 调用 `dewinize` 函数，然后去掉变音符号。

❻ 把德语 Eszett 替换成“ss”（这里没有使用大小写折叠，因为我们想保留大小写）。

❼ 使用 NFKC 规范化形式把字符和与之兼容的码位组合起来。

示例 4-18 是 `asciize` 函数的使用示例。

示例 **4-18**　示例 4-17 中 `asciize` 函数的使用示例

```
>>> order = '“Herr Voß: • ½ cup of OEtker™ caffè latte • bowl of açaí.”'
>>> dewinize(order)
'"Herr Voß: - ½ cup of OEtker(TM) caffè latte - bowl of açaí."'  ❶
>>> asciize(order)
'"Herr Voss: - 1/2 cup of OEtker(TM) caffe latte - bowl of acai."'  ❷
```

❶ `dewinize` 函数替换弯引号、项目符号和™（商标符号）。

❷ `asciize` 函数调用 `dewinize` 函数，去掉变音符号，还会替换 `'ß'`。

不同语言删除变音符号的规则也有所不同。例如，德语把 `'ü'` 变成 `'ue'`。我们定义的 `asciize` 函数没这么精确，因此可能适合你的语言，也可能不适合。不过，它对葡萄牙语的处理是可接受的。

综上，sanitize.py 中的函数做的事情超出了标准的规范化，而且会对文本做进一步处理，很有可能会改变原意。只有知道目标语言、目标用户群和转换后的用途，才能确定要不要做这么深入的规范化。

我们对 Unicode 文本规范化的讨论到此结束。

接下来要解决的 Unicode 问题是……排序。

4.7　Unicode文本排序

Python 比较任何类型的序列时，会一一比较序列里的各个元素。对字符串来说，比较的是码位。可是在比较非 ASCII 字符时，得到的结果不尽如人意。

下面对一个生长在巴西的水果的列表进行排序：

```
>>> fruits = ['caju', 'atemoia', 'cajá', 'açaí', 'acerola']
>>> sorted(fruits)
['acerola', 'atemoia', 'açaí', 'caju', 'cajá']
```

不同的区域采用的排序规则有所不同，葡萄牙语等很多语言按照拉丁字母表排序，重音符号和下加符对排序几乎没什么影响。[9] 因此，排序时“cajá”视作“caja”，必定排在“caju”前面。

> [9]变音符号对排序有影响的情况很少发生，只有两个词之间唯有变音符号不同时才有影响。此时，带有变音符号的词排在常规词的后面。

排序后的 `fruits` 列表应该是：

```
['açaí', 'acerola', 'atemoia', 'cajá', 'caju']
```

在 Python 中，非 ASCII 文本的标准排序方式是使用 `locale.strxfrm` 函数，根据 `locale` 模块的文档（https://docs.python.org/3/library/locale.html?highlight=strxfrm#locale.strxfrm），这 个函数会“把字符串转换成适合所在区域进行比较的形式”。

使用 `locale.strxfrm` 函数之前，必须先为应用设定合适的区域设置，还要祈祷操作系统支持这项设置。在区域设为 `pt_BR` 的 GNU/Linux（Ubuntu 14.04）中，可以使用示例 4-19 中的命令。

示例 **4-19**　使用 `locale.strxfrm` 函数做排序键

```
>>> import locale
>>> locale.setlocale(locale.LC_COLLATE, 'pt_BR.UTF-8')
'pt_BR.UTF-8'
>>> fruits = ['caju', 'atemoia', 'cajá', 'açaí', 'acerola']
>>> sorted_fruits = sorted(fruits, key=locale.strxfrm)
>>> sorted_fruits
['açaí', 'acerola', 'atemoia', 'cajá', 'caju']
```

因此，使用 `locale.strxfrm` 函数做排序键之前，要调用 `setlocale(LC_COLLATE, «your_locale»)`。

不过，有几点要注意。

- 区域设置是全局的，因此不推荐在库中调用 `setlocale` 函数。应用或框架应该在进程启动时设定区域设置，而且此后不要再修改。

- 操作系统必须支持区域设置，否则 `setlocale` 函数会抛出 `locale.Error: unsupported locale setting` 异常。

- 必须知道如何拼写区域名称。它在 Unix 衍生系统中几乎已经形成标准，要通过 `'language_code.encoding'` 获取。[10] 但是在 Windows 中，句法复杂一些：`Language Name-Language Variant_Region Name.codepage`。注意，“Language Name”（语言名称）、“Language Variant”（语言变体）和“Region Name”（区域名）中可以包含空格；除了第一部分之外，其他部分的前面是不同的字符：一个连字符、一个下划线和一个点号。除了语言名称之外，其他部分好像都是可选的。例如，`English_United States.850`，它的语言名称是“English”，区域是“United States”，代码页是“850”。Windows 能理解的语言名称和区域名见于 MSDN 中的文章“Language Identifier Constants and Strings”（https://msdn.microsoft.com/en-us/library/dd318693.aspx），还有“Code Page Identifiers”（https://msdn.microsoft.com/en-us/library/windows/desktop/dd317756(v=vs.85).aspx）一文列出了最后一部分的代码页数字。[11]

- 操作系统的制作者必须正确实现了所设的区域。我在 Ubuntu 14.04 中成功了，但在 OS X（Mavericks 10.9）中却失败了。在两台 Mac 中，调用 `setlocale(LC_COLLATE, 'pt_BR.UTF-8')` 返回的都

是字符串 'pt_BR.UTF-8'，没有任何问题。但是，sorted(fruits, key=locale.strxfrm) 得到的结果与 sorted(fruits) 一样，是错误的。我还在 OS X 中尝试了 fr_FR、es_ES 和 de_DE，但是 locale.strxfrm 并未起作用。[12]

> [10]在 Linux 操作系统中，中国大陆的读者可以使用 zh_CN.UTF-8，简体中文会按照汉语拼音顺序进行排序，它也能对葡萄牙语进行正确排序。——编者注

> [11]感谢 Leonardo Rochael，他所做的工作超出了身为技术审校的职责，虽然他是 GNU/Linux 用户，但却研究了这些 Windows 细节。

> [12]同样，我没找到解决方案，不过却发现其他人也报告了同样的问题。本书技术审校之一 Alex Martelli，在他装有 OS X 10.9 的 Mac 电脑中使用 setlocale 和 locale.strxfrm 时没有遇到问题。综上：结果因人而异。

因此，标准库提供的国际化排序方案可用，但是似乎只支持 GNU/Linux（可能也支持 Windows，但你得是专家）。即便如此，还要依赖区域设置，而这会为部署带来问题。

幸好，有个较为简单的方案：PyPI 中的 PyUCA 库。

使用**Unicode**排序算法排序

James Tauber，一位高产的 Django 贡献者，他一定是感受到了这一痛点，因此开发了 PyUCA 库（https://pypi.python.org/pypi/pyuca/），这是 Unicode 排序算法（Unicode Collation Algorithm，UCA）的纯 Python 实现。示例 4-20 展示了它的简单用法。

示例 **4-20**　使用 pyuca.Collator.sort_key 方法

```
>>> import pyuca
>>> coll = pyuca.Collator()
>>> fruits = ['caju', 'atemoia', 'cajá', 'açaí', 'acerola']
>>> sorted_fruits = sorted(fruits, key=coll.sort_key)
>>> sorted_fruits
['açaí', 'acerola', 'atemoia', 'cajá', 'caju']
```

这样做更友好，而且恰好可用。我在 GNU/Linux、OS X 和 Windows 中[13]

做过测试。目前，PyUCA 只支持 Python 3.*x*。[13]

> [13]2015 年 5 月，PyUCA 重新支持 Python 2.*x*，参见：http://jktauber.com/2015/05/13/pyuca-supports-python-2-again。——编者注

PyUCA 没有考虑区域设置。如果想定制排序方式，可以把自定义的排序表路径传给 `Collator()` 构造方法。PyUCA 默认使用项目自带的 `allkeys.txt`（https://github.com/jtauber/pyuca），这就是 Unicode 6.3.0 的“Default Unicode Collation Element Table”（http://www.unicode.org/Public/UCA/6.3.0/allkeys.txt）的副本。

顺便说一下，那个表是 Unicode 数据库中众多表中的一个。下一节会讨论这个话题。

4.8 Unicode数据库

Unicode 标准提供了一个完整的数据库（许多格式化的文本文件），不仅包括码位与字符名称之间的映射，还有各个字符的元数据，以及字符之间的关系。例如，Unicode 数据库记录了字符是否可以打印、是不是字母、是不是数字，或者是不是其他数值符号。字符串的 `isidentifier`、`isprintable`、`isdecimal` 和 `isnumeric` 等方法就是靠这些信息作判断的。`str.casefold` 方法也用到了 Unicode 表中的信息。

unicodedata 模块中有几个函数用于获取字符的元数据。例如，字符在标准中的官方名称是不是组合字符（如结合波形符构成的变音符号等），以及符号对应的人类可读数值（不是码位）。示例 4-21 展示了 `unicodedata.name()` 和 `unicodedata.numeric()` 函数，以及字符串的 `.isdecimal()` 和 `.isnumeric()` 方法的用法。

示例 **4-21**　Unicode 数据库中数值字符的元数据示例（各个标号说明输出中的各列）

```
import unicodedata
import re

re_digit = re.compile(r'\d')

sample = '1\xbc\xb2\u0969\u136b\u216b\u2466\u2480\u3285'

for char in sample:
    print('U+%04x' % ord(char),                          ❶
          char.center(6),                                ❷
          're_dig' if re_digit.match(char) else '-',     ❸
          'isdig' if char.isdigit() else '-',            ❹
          'isnum' if char.isnumeric() else '-',          ❺
          format(unicodedata.numeric(char), '5.2f'),     ❻
          unicodedata.name(char),                        ❼
          sep='\t')
```

❶ `U+0000` 格式的码位。

❷ 在长度为 6 的字符串中居中显示字符。

❸ 如果字符匹配正则表达式 `r'\d'`，显示 `re_dig`。

❹ 如果 `char.isdigit()` 返回 `True`，显示 `isdig`。

❺ 如果 `char.isnumeric()` 返回 `True`，显示 `isnum`。

❻ 使用长度为 5、小数点后保留 2 位的浮点数显示数值。

❼ Unicode 标准中字符的名称。

运行示例 4-21 得到的结果如图 4-3 所示。

图 4-3 中的第 6 列是在字符上调用 `unicodedata.numeric(char)` 函数得到的结果。这表明，Unicode 知道表示数字的符号的数值。因此，如果你想创建一个支持泰米尔数字和罗马数字的电子表格应用，那就尽管去做吧！

```
$ python3 numerics_demo.py
U+0031    1      re_dig  isdig   isnum    1.00   DIGIT ONE
U+00bc    ¼      -       -       isnum    0.25   VULGAR FRACTION ONE QUARTER
U+00b2    ²      -       isdig   isnum    2.00   SUPERSCRIPT TWO
U+0969    ३      re_dig  isdig   isnum    3.00   DEVANAGARI DIGIT THREE
U+136b    ፫      -       isdig   isnum    3.00   ETHIOPIC DIGIT THREE
U+216b    Ⅻ      -       -       isnum   12.00   ROMAN NUMERAL TWELVE
U+2466    ⑦      -       isdig   isnum    7.00   CIRCLED DIGIT SEVEN
U+2480    ⒀      -       -       isnum   13.00   PARENTHESIZED NUMBER THIRTEEN
U+3285    ㊅      -       -       isnum    6.00   CIRCLED IDEOGRAPH SIX
$
```

图 4-3：9 个数值字符及其元数据；`re_dig` 表示字符匹配正则表达式 `r'\d'`

图 4-3 表明，正则表达式 `r'\d'` 能匹配数字“1”和梵文数字 3，但是不能匹配 `isdigit` 方法判断为数字的其他字符。`re` 模块对 Unicode 的支持并不充分。PyPI 中有个新开发的 `regex` 模块，它的最终目的是取代 `re` 模块，以提供更好的 Unicode 支持。[14] 下一节会回过头来讨论 `re` 模块。

> [14]不过在这个示例中，它在识别数字方面的表现没有 `re` 模块好。

本章使用了 `unicodedata` 模块中的几个函数，但是还有很多没有用到。详情参阅标准库文档对 `unicodedata` 模块的说明（https://docs.python.org/3/library/unicodedata.html）。

在结束对字符串和字节序列的讨论之前，我们还要简要说明一个新的趋势——双模式 API，即提供的函数能接受字符串或字节序列为参数，然后根据类型进行特殊处理。

4.9　支持字符串和字节序列的双模式API

标准库中的一些函数能接受字符串或字节序列为参数，然后根据类型展现不同的行为。`re` 和 `os` 模块中就有这样的函数。

4.9.1　正则表达式中的字符串和字节序列

如果使用字节序列构建正则表达式，`\d` 和 `\w` 等模式只能匹配 ASCII 字符；相比之下，如果是字符串模式，就能匹配 ASCII 之外的 Unicode 数字或字母。示例 4-22 和图 4-4 展示了字符串模式和字节序列模式中字母、ASCII 数字、上标和泰米尔数字的匹配情况。

示例 **4-22**　ramanujan.py：比较简单的字符串正则表达式和字节序列正则表达式的行为

```
import re

re_numbers_str = re.compile(r'\d+')     ❶
re_words_str = re.compile(r'\w+')
re_numbers_bytes = re.compile(rb'\d+')  ❷
re_words_bytes = re.compile(rb'\w+')

text_str = ("Ramanujan saw \u0be7\u0bed\u0be8\u0bef"  ❸
            " as 1729 = 1³ + 12³ = 9³ + 10³.")        ❹

text_bytes = text_str.encode('utf_8')  ❺

print('Text', repr(text_str), sep='\n  ')
print('Numbers')
print('  str  :', re_numbers_str.findall(text_str))      ❻
print('  bytes:', re_numbers_bytes.findall(text_bytes))  ❼
print('Words')
print('  str  :', re_words_str.findall(text_str))        ❽
print('  bytes:', re_words_bytes.findall(text_bytes))    ❾
```

❶ 前两个正则表达式是字符串类型。

❷ 后两个正则表达式是字节序列类型。

❸ 要搜索的 Unicode 文本，包括 1729 的泰米尔数字（逻辑行直到右括号才结束）。

❹ 这个字符串在编译时与前一个拼接起来（参见 Python 语言参考手册中的“2.4.2. String literal concatenation”，https://docs.python.org/3/reference/lexical_analysis.html#stri
literal-concatenation）。

❺ 字节序列只能用字节序列正则表达式搜索。

❻ 字符串模式 `r'\d+'` 能匹配泰米尔数字和 ASCII 数字。

❼ 字节序列模式 `rb'\d+'` 只能匹配 ASCII 字节中的数字。

❽ 字符串模式 `r'\w+'` 能匹配字母、上标、泰米尔数字和 ASCII 数字。

❾ 字节序列模式 `rb'\w+'` 只能匹配 ASCII 字节中的字母和数字。

```
$ python3 ramanujan.py
Text
  'Ramanujan saw ௧௭௨௯ as 1729 = 1³ + 12³ = 9³ + 10³.'
Numbers
  str  : ['௧௭௨௯', '1729', '1', '12', '9', '10']
  bytes: [b'1729', b'1', b'12', b'9', b'10']
Words
  str  : ['Ramanujan', 'saw', '௧௭௨௯', 'as', '1729', '1³', '12³', '9³', '10³']
  bytes: [b'Ramanujan', b'saw', b'as', b'1729', b'1', b'12', b'9', b'10']
$
```

图 **4-4**：运行示例 **4-22** 中的 **ramanujan.py** 脚本时的截图

示例 4-22 是随便举的例子，为的是说明一个问题：可以使用正则表达式搜索字符串和字节序列，但是在后一种情况中，ASCII 范围外的字节不会当成数字和组成单词的字母。

字符串正则表达式有个 `re.ASCII` 标志，它让 `\w`、`\W`、`\b`、`\B`、`\d`、`\D`、`\s` 和 `\S` 只匹配 ASCII 字符。详情参阅 `re` 模块的文档（https://docs.python.org/3/library/re.html）。

另一个重要的双模式模块是 os。

4.9.2　os函数中的字符串和字节序列

GNU/Linux 内核不理解 Unicode，因此你可能发现了，对任何合理的编码方案来说，在文件名中使用字节序列都是无效的，无法解码成字符串。在不同操作系统中使用各种客户端的文件服务器，在遇到这个问题时尤其容易出错。

为了规避这个问题，os 模块中的所有函数、文件名或路径名参数既能使用字符串，也能使用字节序列。如果这样的函数使用字符串参数调用，该参数会使用 sys.getfilesystemencoding() 得到的编解码器自动编码，然后操作系统会使用相同的编解码器解码。这几乎就是我们想要的行为，与 Unicode 三明治最佳实践一致。

但是，如果必须处理（也可能是修正）那些无法使用上述方式自动处理的文件名，可以把字节序列参数传给 os 模块中的函数，得到字节序列返回值。这一特性允许我们处理任何文件名或路径名，不管里面有多少鬼符，如示例 4-23 所示。

示例 **4-23**　把字符串和字节序列参数传给 listdir 函数得到的结果

```
>>> os.listdir('.') #  ❶
['abc.txt', 'digits-of-π.txt']
>>> os.listdir(b'.') #  ❷
[b'abc.txt', b'digits-of-\xcf\x80.txt']
```

❶ 第二个文件名是“digits-of-π.txt”（有一个希腊字母 π）。

❷ 参数是字节序列，listdir 函数返回的文件名也是字节序列：b'\xcf\x80' 是希腊字母 π 的 UTF-8 编码。

为了便于手动处理字符串或字节序列形式的文件名或路径名，os 模块提供了特殊的编码和解码函数。

fsencode(filename)

如果 `filename` 是 `str` 类型（此外还可能是 `bytes` 类型），使用 `sys.getfilesystemencoding()` 返回的编解码器把 `filename` 编码成字节序列；否则，返回未经修改的 `filename` 字节序列。

`fsdecode(filename)`

如果 `filename` 是 `bytes` 类型（此外还可能是 `str` 类型），使用 `sys.getfilesystemencoding()` 返回的编解码器把 `filename` 解码成字符串；否则，返回未经修改的 `filename` 字符串。

在 Unix 衍生平台中，这些函数使用 `surrogateescape` 错误处理方式（参见下述附注栏）以避免遇到意外字节序列时卡住。Windows 使用的错误处理方式是 `strict`。

使用 **`surrogateescape`** 处理鬼符

Python 3.1 引入的 `surrogateescape` 编解码器错误处理方式是处理意外字节序列或未知编码的一种方式，它的说明参见“PEP 383 — Non-decodable Bytes in System Character Interfaces”（https://www.python.org/dev/peps/pep-0383/）。

这种错误处理方式会把每个无法解码的字节替换成 Unicode 中 U+DC00 到 U+DCFF 之间的码位（Unicode 标准把这些码位称为“Low Surrogate Area”），这些码位是保留的，没有分配字符，供应用程序内部使用。编码时，这些码位会转换成被替换的字节值，如示例 4-24 所示。

示例 **4-24**　使用 `surrogateescape` 错误处理方式

```
>>> os.listdir('.')  ❶
['abc.txt', 'digits-of-π.txt']
>>> os.listdir(b'.')  ❷
[b'abc.txt', b'digits-of-\xcf\x80.txt']
>>> pi_name_bytes = os.listdir(b'.')[1]  ❸
>>> pi_name_str = pi_name_bytes.decode('ascii', 'surrogateescape')  ❹
>>> pi_name_str  ❺
'digits-of-\udccf\udc80.txt'
>>> pi_name_str.encode('ascii', 'surrogateescape')  ❻
b'digits-of-\xcf\x80.txt
```

❶ 列出目录里的文件，有个文件名中包含非 ASCII 字符。

❷ 假设我们不知道编码，获取文件名的字节序列形式。

❸ `pi_names_bytes` 是包含 π 的文件名。

❹ 使用`'ascii'` 编解码器和 `'surrogateescape'` 错误处理方式把它解码成字符串。

❺ 各个非 ASCII 字节替换成代替码位：`'\xcf\x80'` 变成了`'\udccf\udc80'`。

❻ 编码成 ASCII 字节序列：各个代替码位还原成被替换的字节。

我们对字符串和字节序列的探讨到此结束。如果你坚持读到这里，恭喜你！

4.10 本章小结

本章首先澄清了人们对一个字符等于一个字节的误解。随着 Unicode 的广泛使用（80% 的网站已经使用 UTF-8），我们必须把文本字符串与它们在文件中的二进制序列表述区分开，而 Python 3 中这个区分是强制的。

对 `bytes`、`bytearray` 和 `memoryview` 等二进制序列数据类型做了简要概述之后，我们转到了编码和解码话题，通过示例展示了重要的编解码器；随后讨论了如何避免和处理臭名昭著的 `UnicodeEncodeError` 和 `UnicodeDecodeError`，以及由于 Python 源码文件编码错误导致的 `SyntaxError`。

讨论源码的编码问题时，我表明了自己对非 ASCII 标识符的观点：如果代码基的维护者想使用包含非 ASCII 字符的人类语言命名标识符，那就去做，除非还想在 Python 2 中运行代码。但是，如果项目想吸引世界各国的贡献者，那么标识符应该使用英语单词，此时 ASCII 就够用了。

然后，我们说明了在没有元数据的情况下检测编码的理论和实际情况：理论上，做不到这一点；但是实际上，Chardet 包能够正确处理一些流行的编码。随后介绍了字节序标记，这是 UTF-16 和 UTF-32 文件中常见的编码提示，某些 UTF-8 文件中也有。

随后的一节演示了如何打开文本文件，这是一项简单的任务，不过有个陷阱：打开文本文件时，`encoding=` 关键字参数不是必需的，但是应该指定。如果没有指定编码，那么程序会想方设法生成“纯文本”，如此一来，不一致的默认编码就会导致跨平台不兼容性。然后，我们说明了 Python 用作默认值的几个编码设置，以及如何检测它们：`locale.getpreferredencoding()`、`sys.getfilesystemencod` 以及标准 I/O 文件（如 `sys.stdout.encoding`）的编码。对 Windows 用户来说，现实不容乐观：这些设置在同一台设备中往往有不同的值，而且各个设置相互不兼容。而对 GNU/ Linux 和 OS X 用户来说，情况就好多了，几乎所有地方使用的默认值都是 `UTF-8`。

文本比较是个异常复杂的任务，因为 Unicode 为某些字符提供了不同的

表示，所以匹配文本之前一定要先规范化。说明规范化和大小写折叠之后，我们提供了几个实用函数，你可以根据自己的需求改编。其中有个函数所做的是极端转换，比如去掉所有重音符号。随后，我们说明了如何使用标准库中的 `locale` 模块正确地排序 Unicode 文本（有一些注意事项）；此外，还可以使用外部的 PyUCA 包，从而无需依赖捉摸不定的区域配置。

最后简要介绍了 Unicode 数据库（包含每个字符的元数据），还简单讨论了双模式 API（例如 `re` 和 `os` 模块，这两个模块中的某些函数可以接受字符串或字节序列参数，返回不同但合适的结果）。

4.11　延伸阅读

Ned Batchelder 在 2012 年的 PyCon US 所做的演讲“Pragmatic Unicode—or—How Do I Stop the Pain?”（http://nedbatchelder.com/text/unipain.html）非常出色。Ned 很专业，除了幻灯片和视频之外，他还提供了完整的文字记录。Esther Nam 和 Travis Fischer 在 PyCon 2014 做了一场精彩的演讲：“Character encoding and Unicode in Python: How to (╯°□°)╯︵ ┻━┻ with dignity”[幻灯片（http://www.slideshare.net/fischertrav/character-encoding-unicode-how-to-with-dignity-33352863），视频（http://pyvideo.org/pycon-us-2014/character-encoding-and-unicode-in-python.html）]。本章开头那句简短有力的话就是出自这次演讲：“人类使用文本，计算机使用字节序列。”本书的技术审校之一 Lennart Regebro 在“Unconfusing Unicode: What Is Unicode?”（https://regebro.wordpress.com/2011/03/23/unconfusing-unicode-what-is-unicode/）这篇短文中提出了“Useful Mental Model of Unicode（UMMU）”。Unicode 是个复杂的标准，Lennart 提出的 UMMU 是个很好的切入点。

Python 文档中的“Unicode HOWTO”一文（https://docs.python.org/3/howto/unicode.html）从几个不同的角度对本章所涉及的话题做了讨论，从编码历史到句法细节、编解码器、正则表达式、文件名和 Unicode 的 I/O 最佳实践（即 Unicode 三明治），而且各节都给出了大量参考资料链接。*Dive into Python 3* 是一本非常优秀的书（Mark Pilgrim 著，http://www.diveintopython3.net），其中第 4 章“Strings”（http://www.diveintopython3.net/strings.html）对 Python 3 对 Unicode 的支持做了很好的介绍。此外，该书的第 15 章（http://getpython3.com/diveintopython3/case-study-porting-chardet-to-python-3.html）阐述了 Chardet 库从 Python 2 移植到 Python 3 的过程，这是一个宝贵的案例分析，从中可以看出，从旧的 `str` 类型转到新的 `bytes` 类型是造成迁移如此痛苦的主要原因，也是检测编码的库应该关注的重点。

如果你用过 Python 2，但是刚接触 Python 3，可以阅读 Guido van Rossum 写的“What's New in Python 3.0”（https://docs.python.org/3.0/whatsnew/3.0.html#text-vs-data-instead-of-

unicode-vs-8-bit），这篇文章简要列出了新版的 15 点变化，而且附有很多链接。Guido 开门见山地说道："你自以为知道的二进制数据和 Unicode 知识全都变了。"Armin Ronacher 的博客文章"The Updated Guide to Unicode on Python"（http://lucumr.pocoo.org/2013/7/2/the-updated-guide-to-unicode/）深入分析了 Python 3 中 Unicode 的一些陷阱（Armin 不是很热衷于 Python 3）。

《Python Cookbook（第 3 版）中文版》（David Beazley 和 Brian K. Jones 著）的第 2 章"字符串和文本"中有几个诀窍谈到了 Unicode 规范化、清洗文本，以及在字节序列上执行面向文本的操作。第 5 章涵盖文件和 I/O，"5.17 将字节数据写入文本文件"指出，任何文本文件的底层都有一个二进制流，如果需要可以直接访问。之后的"6.11 读写二进制结构的数组"用到了 `struct` 模块。

Nick Coghlan 的"Python Notes"博客中有两篇文章与本章的话题十分相关："Python 3 and ASCII Compatible Binary Protocols"（http://python-notes.curiousefficiency.org/en/latest/python3/binary_protocols.html）和"Processing Text Files in Python 3"（http://python-notes.curiousefficiency.org/en/latest/python3/text_file_processing.html）。强烈推荐阅读。

Python 3.5 将为二进制序列引入新的构造方法和方法，而且会废弃目前使用的构造方法签名（参见"PEP 467—Minor API improvements for binary sequences"，https://www.python.org/dev/peps/pep-0467/）。此外，Python 3.5 还会实现"PEP 461—Adding % formatting to bytes and bytearray"（https://www.python.org/dev/peps/pep-0461/）。

Python 支持的编码列表参见 `codecs` 模块文档的"Standard Encodings"一节（https://docs.python.org/3/library/codecs.html#standard-encodings）。如果需要通过编程的方式获得那个列表，看看 CPython 源码中 /Tools/unicode/listcodecs.py 脚本（https://hg.python.org/cpython/file/6dcc96fa3970/Tools/unicode/listcodecs.py）是怎么做的。

Martijn Faassen 的文章"Changing the Python Default Encoding Considered Harmful"（http://blog.startifact.com/posts/older/changing-the-python-default-encoding-considered-harmful.html）和 Tarek Ziadé的文章"sys.setdefaultencoding Is

Evil”（http://blog.ziade.org/2008/01/08/syssetdefaultencoding-is-evil/）解释了为什么一定不能修改 `sys.getdefaultencoding()` 获取的编码，即便知道怎么做也不能改。

Unicode Explained（Jukka K. Korpela 著，O'Reilly 出版社，http://shop.oreilly.com/product/9780596101213.do）和 *Unicode Demystified*（Richard Gillam 著，Addison-Wesley 出版社，http://www.informit.com/store/unicode-demystified-a-practical-programmers-guide-to-9780201700527）这两本书不是针对 Python 的，但在我学习 Unicode 相关概念时给了我很大的帮助。Victor Stinner 的著作 *Programming with Unicode*（http://unicodebook.readthedocs.org/index.html）是一本免费的自出版图书（遵守 CC BY-SA 协议），其中讨论了一般的 Unicode 话题，以及主流操作系统和几门编程语言（包括 Python）中的相关工具和 API。

W3C 网站中的“Case Folding: An Introduction”（https://www.w3.org/International/wiki/Case_folding）和“Character Model for the World Wide Web: String Matching and Searching”（https://www.w3.org/TR/charmod-norm/）讨论了规范化相关的概念，前者是介绍性文章，后者则是以枯燥的标准用语写就的工作草案——“Unicode Standard Annex #15—Unicode Normalization Forms”（http://unicode.org/reports/tr15/）也是这种风格。Unicode.org 网站中的“Frequently Asked Questions / Normalization”（http://www.unicode.org/faq/normalization.html）更容易理解， Mark Davis 写的“NFC FAQ”（http://www.macchiato.com/unicode/nfc-faq）也是如此。Mark 是多个 Unicode 算法的作者，在我写作本书时，他还担任 Unicode 联盟的主席。

杂谈

“纯文本”是什么

对于经常处理非英语文本的人来说，“纯文本”并不是指“ASCII”。Unicode 词汇表（http://www.unicode.org/glossary/#plain_text）是这样定义纯文本的：

> 只由特定标准的码位序列组成的计算机编码文本，其中不含其

他格式化或结构化信息。

这个定义的前半句说得很好，但是我不同意后半句。HTML 就是包含格式化和结构化信息的纯文本格式，但它依然是纯文本，因为 HTML 文件中的每个字节都表示文本字符（通常使用 UTF-8 编码），没有任何字节表示文本之外的信息。.png 或 .xsl 文档则不同，其中多数字节表示打包的二进制值，例如 RGB 值和浮点数。在纯文本中，数字使用数字符号序列表示。

这本书是我用一种名为 AsciiDoc（http://www.methods.co.nz/asciidoc/，很讽刺）的纯文本格式撰写的，它是 O'Reilly 优秀的图书出版平台 Atlas（https://atlas.oreilly.com/）的工具链中的一部分。AsciiDoc 的源文件是纯文本，但用的是 UTF-8 编码，而不是 ASCII。如果不这样做的话，撰写本章必定痛苦不堪。姑且不管名称，AsciiDoc 是个很棒的工具。

Unicode 的世界正在不断扩张，但是有些边缘场景缺少支持工具。因此图 4-1、图 4-3 和图 4-4 中的内容要使用图像，因为渲染本书的字体中缺少一些我想展示的字符。不过，Ubuntu 14.04 和 OS X 10.9 的终端能正确显示，包括“mojibake”（文字化け）这个日文的词。

捉摸不透的 **Unicode**

讨论 Unicode 规范化时，我经常使用“往往”“多数”和“通常”等不确定的修饰语。很遗憾，我不能提供更可靠的建议，因为 Unicode 规则有很多例外，很难百分之百确定。

例如，µ（微符号）是“兼容字符”，而 Ω（欧姆）和 Å（埃）符号却不是。这种差别是有真实影响的：NFC 规范化形式（推荐用于文本匹配）会把 Ω（欧姆）替换成 Ω（大写希腊字母欧米加），把 Å（埃）替换成 Å（上有圆圈的大写字母 A）。但是，作为“兼容字符”的 µ（微符号）不会替换成视觉等效的 μ（小写希腊字母 μ）；不过在使用更极端的 NFKC 或 NFKD 规范化形式时会替换，但这是有损转换。

我能理解为什么把 µ（微符号）纳入 Unicode，因为 `latin1` 编码中

有它，如果换成希腊字母 μ，会破坏两种编码之间的转换。说到底，这就是微符号是“兼容字符”的原因。但是，如果是由于兼容原因而没把欧姆和埃符号纳入 Unicode，那为什么这两个符号要存在？ Unicode 已经为 `GREEK CAPITAL LETTER OMEGA` 和 `LATIN CAPITAL LETTER A WITH RING ABOVE` 分配了码位，它们的外观一样，而且 NFC 规范化形式会替换它们。想想看吧。

研究 Unicode 几小时之后，我猜测的原因是：Unicode 异常复杂，充满特殊情况，而且要覆盖各种人类语言和产业标准策略。

在 **RAM** 中如何表示字符串

Python 官方文档对字符串的码位在内存中如何存储避而不谈。毕竟，这是实现细节。理论上，怎么存储都没关系：不管内部表述如何，输出时每个字符串都要编码成字节序列。

在内存中，Python 3 使用固定数量的字节存储字符串的各个码位，以便高效访问各个字符或切片。

在 Python 3.3 之前，编译 CPython 时可以配置在内存中使用 16 位或 32 位存储各个码位。16 位是“窄构建”（narrow build），32 位是“宽构建”（wide build）。如果想知道用的是哪个，要查看 `sys.maxunicode` 的值：65535 表示“窄构建”，不能透明地处理 U+FFFF 以上的码位。“宽构建”没有这个限制，但是消耗的内存更多：每个字符占 4 个字节，就算是中文象形文字的码位大多数也只占 2 个字节。这两种构建没有高下之分，应该根据自己的需求选择。

从 Python 3.3 起，创建 `str` 对象时，解释器会检查里面的字符，然后为该字符串选择最经济的内存布局：如果字符都在 `latin1` 字符集中，那就使用 1 个字节存储每个码位；否则，根据字符串中的具体字符，选择 2 个或 4 个字节存储每个码位。这是简述，完整细节参阅“PEP 393—Flexible String Representation”（https://www.python.org/dev/peps/pep-0393/）。

灵活的字符串表述类似于 Python 3 对 `int` 类型的处理方式：如果一个整数在一个机器字中放得下，那就存储在一个机器字中；否则解释器切换成变长表述，类似于 Python 2 中的 `long` 类型。这种聪明

的做法得到推广，真是让人欢喜！

第三部分　把函数视作对象

第 5 章　一等函数

> 不管别人怎么说或怎么想，我从未觉得 Python 受到来自函数式语言的太多影响。我非常熟悉命令式语言，如 C 和 Algol 68，虽然我把函数定为一等对象，但是我并不把 Python 当作函数式编程语言。[1]
>
> ——Guido van Rossum
> Python 仁慈的独裁者

[1]摘录自 Guido 的 The History of Python 博客，“Origins of Python's Functional Features”（http://python-history.blogspot.jp/2009/04/origins-of-pythons-functional-features.html）。

在 Python 中，函数是一等对象。编程语言理论家把“一等对象”定义为满足下述条件的程序实体：

- 在运行时创建
- 能赋值给变量或数据结构中的元素
- 能作为参数传给函数
- 能作为函数的返回结果

在 Python 中，整数、字符串和字典都是一等对象——没什么特别的。如果在 Python 之前，你使用的语言并未把函数当作一等公民，那么本章以及第三部分余下的内容将重点讨论把函数作为对象的影响和实际应用。

> 人们经常将“把函数视作一等对象”简称为“一等函数”。这样说并不完美，似乎表明这是函数中的特殊群体。在 Python 中，所有函数都是一等对象。

5.1　把函数视作对象

示例 5-1 中的控制台会话表明，Python 函数是对象。这里我们创建了一个函数，然后调用它，读取它的 `__doc__` 属性，并且确定函数对象本身是 `function` 类的实例。

示例 **5-1**　创建并测试一个函数，然后读取它的 `__doc__` 属性，再检查它的类型

```
>>> def factorial(n):  ➊
...     '''returns n!'''
...     return 1 if n < 2 else n * factorial(n-1)
...
>>> factorial(42)
1405006117752879898543142606244511569936384000000000
>>> factorial.__doc__  ➋
'returns n!'
>>> type(factorial)  ➌
<class 'function'>
```

➊ 这是一个控制台会话，因此我们是在“运行时”创建一个函数。

➋ `__doc__` 是函数对象众多属性中的一个。

➌ `factorial` 是 `function` 类的实例。

`__doc__` 属性用于生成对象的帮助文本。在 Python 交互式控制台中，`help(factorial)` 命令输出的内容如图 5-1 所示。

```
1. less
Help on function factorial in module __main__:

factorial(n)
    returns n!
~
~
(END)
```

图 **5-1**：**`factorial`** 函数的帮助界面；输出的文本来自函数对象的 **`__doc__`** 属性

示例 5-2 展示了函数对象的"一等"本性。我们可以把 `factorial` 函数赋值给变量 `fact`，然后通过变量名调用。我们还能把它作为参数传给 `map` 函数。`map` 函数返回一个可迭代对象，里面的元素是把第一个参数（一个函数）应用到第二个参数（一个可迭代对象，这里是 `range(11)`）中各个元素上得到的结果。

示例 **5-2**　通过别的名称使用函数，再把函数作为参数传递

```
>>> fact = factorial
>>> fact
<function factorial at 0x...>
>>> fact(5)
120
>>> map(factorial, range(11))
<map object at 0x...>
>>> list(map(fact, range(11)))
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

有了一等函数，就可以使用函数式风格编程。函数式编程的特点之一是使用高阶函数——这是下一节的话题。

5.2　高阶函数

接受函数为参数，或者把函数作为结果返回的函数是高阶函数（higher-order function）。`map` 函数就是一例，如示例 5-2 所示。此外，内置函数 `sorted` 也是：可选的 `key` 参数用于提供一个函数，它会应用到各个元素上进行排序，参见 2.7 节。

例如，若想根据单词的长度排序，只需把 `len` 函数传给 `key` 参数，如示例 5-3 所示。

示例 **5-3**　根据单词长度给一个列表排序

```
>>> fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry', 'banana']
>>> sorted(fruits, key=len)
['fig', 'apple', 'cherry', 'banana', 'raspberry', 'strawberry']
>>>
```

任何单参数函数都能作为 `key` 参数的值。例如，为了创建押韵词典，可以把各个单词反过来拼写，然后排序。注意，示例 5-4 中列表里的单词没有变，我们只是把反向拼写当作排序条件，因此各种浆果（berry）都排在一起。

示例 **5-4**　根据反向拼写给一个单词列表排序

```
>>> def reverse(word):
...     return word[::-1]
>>> reverse('testing')
'gnitset'
>>> sorted(fruits, key=reverse)
['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']
>>>
```

在函数式编程范式中，最为人熟知的高阶函数有 `map`、`filter`、`reduce` 和 `apply`。`apply` 函数在 Python 2.3 中标记为过时，在 Python 3 中移除了，因为不再需要它了。如果想使用不定量的参数调用函数，可以编写 `fn(*args, **keywords)`，不用再编写 `apply(fn, args, kwargs)`。

map、filter 和 reduce 这三个高阶函数还能见到，不过多数使用场景下都有更好的替代品。详情参阅下一节。

map、filter和reduce的现代替代品

函数式语言通常会提供 map、filter 和 reduce 三个高阶函数（有时使用不同的名称）。在 Python 3 中，map 和 filter 还是内置函数，但是由于引入了列表推导和生成器表达式，它们变得没那么重要了。列表推导或生成器表达式具有 map 和 filter 两个函数的功能，而且更易于阅读，如示例 5-5 所示。

示例 **5-5**　计算阶乘列表：map 和 filter 与列表推导比较

```
>>> list(map(fact, range(6)))  ❶
[1, 1, 2, 6, 24, 120]
>>> [fact(n) for n in range(6)]  ❷
[1, 1, 2, 6, 24, 120]
>>> list(map(factorial, filter(lambda n: n % 2, range(6))))  ❸
[1, 6, 120]
>>> [factorial(n) for n in range(6) if n % 2]  ❹
[1, 6, 120]
>>>
```

❶ 构建 0! 到 5! 的一个阶乘列表。

❷ 使用列表推导执行相同的操作。

❸ 使用 map 和 filter 计算直到 5! 的奇数阶乘列表。

❹ 使用列表推导做相同的工作，换掉 map 和 filter，并避免了使用 lambda 表达式。

在 Python 3 中，map 和 filter 返回生成器（一种迭代器），因此现在它们的直接替代品是生成器表达式（在 Python 2 中，这两个函数返回列表，因此最接近的替代品是列表推导）。

在 Python 2 中，reduce 是内置函数，但是在 Python 3 中放到 functools 模块里了。这个函数最常用于求和，自 2003 年发布的 Python 2.3 开始，最好使用内置的 sum 函数。在可读性和性能方面，这

是一项重大改善（见示例 5-6）。

示例 **5-6**　使用 `reduce` 和 `sum` 计算 0~99 之和

```
>>> from functools import reduce  ❶
>>> from operator import add ❷
>>> reduce(add, range(100))  ❸
4950
>>> sum(range(100))  ❹
4950
>>>
```

❶ 从 Python 3.0 起，`reduce` 不再是内置函数了。

❷ 导入 `add`，以免创建一个专求两数之和的函数。

❸ 计算 0~99 之和。

❹ 使用 `sum` 做相同的求和；无需导入或创建求和函数。

`sum` 和 `reduce` 的通用思想是把某个操作连续应用到序列的元素上，累计之前的结果，把一系列值归约成一个值。

`all` 和 `any` 也是内置的归约函数。

`all(iterable)`

　　如果 `iterable` 的每个元素都是真值，返回 `True`；`all([])` 返回 `True`。

`any(iterable)`

　　只要 `iterable` 中有元素是真值，就返回 `True`；`any([])` 返回 `False`。

10.6 节将深入说明 `reduce` 函数，我会不断改进一个示例，为这个函数提供有意义的上下文。本书后面的 14.11 节将重点讨论可迭代对象，届时会概述各个归约函数。

为了使用高阶函数，有时创建一次性的小型函数更便利。这便是匿名函数存在的原因，下一节将会讨论。

5.3　匿名函数

`lambda` 关键字在 Python 表达式内创建匿名函数。

然而，Python 简单的句法限制了 `lambda` 函数的定义体只能使用纯表达式。换句话说，`lambda` 函数的定义体中不能赋值，也不能使用 `while` 和 `try` 等 Python 语句。

在参数列表中最适合使用匿名函数。例如，示例 5-7 使用 `lambda` 表达式重写了示例 5-4 中排序押韵单词的示例，这样就省掉了 `reverse` 函数。

示例 **5-7**　使用 `lambda` 表达式反转拼写，然后依此给单词列表排序

```
>>> fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry', 'banana']
>>> sorted(fruits, key=lambda word: word[::-1])
['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']
>>>
```

除了作为参数传给高阶函数之外，Python 很少使用匿名函数。由于句法上的限制，非平凡的 `lambda` 表达式要么难以阅读，要么无法写出。

Lundh 提出的 `lambda` 表达式重构秘笈

如果使用 `lambda` 表达式导致一段代码难以理解，Fredrik Lundh 建议像下面这样重构。

(1) 编写注释，说明 `lambda` 表达式的作用。

(2) 研究一会儿注释，并找出一个名称来概括注释。

(3) 把 `lambda` 表达式转换成 `def` 语句，使用那个名称来定义函数。

(4) 删除注释。

这几步摘自“Functional Programming HOWTO”（https://docs.python.org/3/howto/functional.html），这是一篇必读文章。

`lambda` 句法只是语法糖：与 `def` 语句一样，`lambda` 表达式会创建函数对象。这是 Python 中几种可调用对象的一种。下一节会说明所有可调用对象。

5.4 可调用对象

除了用户定义的函数，调用运算符（即 `()`）还可以应用到其他对象上。如果想判断对象能否调用，可以使用内置的 `callable()` 函数。Python 数据模型文档列出了 7 种可调用对象。

用户定义的函数

　　使用 `def` 语句或 `lambda` 表达式创建。

内置函数

　　使用 C 语言（CPython）实现的函数，如 `len` 或 `time.strftime`。

内置方法

　　使用 C 语言实现的方法，如 `dict.get`。

方法

　　在类的定义体中定义的函数。

类

　　调用类时会运行类的 `__new__` 方法创建一个实例，然后运行 `__init__` 方法，初始化实例，最后把实例返回给调用方。因为 Python 没有 `new` 运算符，所以调用类相当于调用函数。（通常，调用类会创建那个类的实例，不过覆盖 `__new__` 方法的话，也可能出现其他行为。19.1.3 节会见到一个例子。）

类的实例

　　如果类定义了 `__call__` 方法，那么它的实例可以作为函数调用。参见 5.5 节。

生成器函数

使用 `yield` 关键字的函数或方法。调用生成器函数返回的是生成器对象。

生成器函数在很多方面与其他可调用对象不同，详情参见第 14 章。生成器函数还可以作为协程，参见第 16 章。



Python 中有各种各样可调用的类型，因此判断对象能否调用，最安全的方法是使用内置的 `callable()` 函数：

```
>>> abs, str, 13
(<built-in function abs>, <class 'str'>, 13)
>>> [callable(obj) for obj in (abs, str, 13)]
[True, True, False]
```

接下来说明如何把类的实例变成可调用的对象。

5.5　用户定义的可调用类型

不仅 Python 函数是真正的对象，任何 Python 对象都可以表现得像函数。为此，只需实现实例方法 `__call__`。

示例 5-8 实现了 `BingoCage` 类。这个类的实例使用任何可迭代对象构建，而且会在内部存储一个随机顺序排列的列表。调用实例会取出一个元素。

示例 **5-8**　bingocall.py：调用 `BingoCage` 实例，从打乱的列表中取出一个元素

```
import random

class BingoCage:

def __init__(self, items):
    self._items = list(items)  ❶
    random.shuffle(self._items)  ❷

def pick(self):  ❸
    try:
        return self._items.pop()
    except IndexError:
        raise LookupError('pick from empty BingoCage')  ❹

def __call__(self):  ❺
    return self.pick()
```

❶ `__init__` 接受任何可迭代对象；在本地构建一个副本，防止列表参数的意外副作用。

❷ `shuffle` 定能完成工作，因为 `self._items` 是列表。

❸ 起主要作用的方法。

❹ 如果 `self._items` 为空，抛出异常，并设定错误消息。

❺ bingo.pick() 的快捷方式是 bingo()。

下面是示例 5-8 中定义的类的简单演示。注意，bingo 实例可以作为函数调用，而且内置的 callable(...) 函数判定它是可调用的对象：

```
>>> bingo = BingoCage(range(3))
>>> bingo.pick()
1
>>> bingo()
0
>>> callable(bingo)
True
```

实现 __call__ 方法的类是创建函数类对象的简便方式，此时必须在内部维护一个状态，让它在调用之间可用，例如 BingoCage 中的剩余元素。装饰器就是这样。装饰器必须是函数，而且有时要在多次调用之间“记住”某些事 [例如备忘（memoization），即缓存消耗大的计算结果，供后面使用]。

创建保有内部状态的函数，还有一种截然不同的方式——使用闭包。闭包和装饰器在第 7 章讨论。

下面讨论把函数视作对象处理的另一方面：运行时内省。

5.6　函数内省

除了 `__doc__`，函数对象还有很多属性。使用 `dir` 函数可以探知 `factorial` 具有下述属性：

```
>>> dir(factorial)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
'__defaults__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__get__', '__getattribute__', '__globals__',
'__gt__', '__hash__', '__init__', '__kwdefaults__', '__le__', '__lt__',
'__module__', '__name__', '__ne__', '__new__', '__qualname__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__']
>>>
```

其中大多数属性是 Python 对象共有的。本节讨论与把函数视作对象相关的几个属性，先从 `__dict__` 开始。

与用户定义的常规类一样，函数使用 `__dict__` 属性存储赋予它的用户属性。这相当于一种基本形式的注解。一般来说，为函数随意赋予属性不是很常见的做法，但是 Django 框架这么做了。参见“The Django admin site”文档（https://docs.djangoproject.com/en/1.10/ref/contrib/admin/）中对 `short_description`、`boolean` 和 `allow_tags` 属性的说明。这篇 Django 文档中举了下述示例，把 `short_description` 属性赋予一个方法，Django 管理后台使用这个方法时，在记录列表中会出现指定的描述文本：

```
    def upper_case_name(obj):
        return ("%s %s" % (obj.first_name, obj.last_name)).upper()
    upper_case_name.short_description = 'Customer name'
```

下面重点说明函数专有而用户定义的一般对象没有的属性。计算两个属性集合的差集便能得到函数专有属性列表（见示例 5-9）。

示例 **5-9**　列出常规对象没有而函数有的属性

```
>>> class C: pass  # ❶
>>> obj = C()  # ❷
>>> def func(): pass  # ❸
>>> sorted(set(dir(func)) - set(dir(obj))) # ❹
['__annotations__', '__call__', '__closure__', '__code__', '__defaults__',
'__get__', '__globals__', '__kwdefaults__', '__name__', '__qualname__']
>>>
```

❶ 创建一个空的用户定义的类。

❷ 创建一个实例。

❸ 创建一个空函数。

❹ 计算差集，然后排序，得到类的实例没有而函数有的属性列表。

表 5-1 对示例 5-9 中列出的属性做了简要说明。

表**5-1**：用户定义的函数的属性

名称	类型	说明
`__annotations__`	`dict`	参数和返回值的注解
`__call__`	`method-wrapper`	实现 `()` 运算符；即可调用对象协议
`__closure__`	`tuple`	函数闭包，即自由变量的绑定（通常是 `None`）
`__code__`	`code`	编译成字节码的函数元数据和函数定义体
`__defaults__`	`tuple`	形式参数的默认值
`__get__`	`method-wrapper`	实现只读描述符协议（参见第 20 章）

__globals__	dict	函数所在模块中的全局变量
__kwdefaults__	dict	仅限关键字形式参数的默认值
__name__	str	函数名称
__qualname__	str	函数的限定名称，如 Random.choice（参阅PEP 3155，https://www.python.org/dev/peps/pep-3155/）

后面几节会讨论 __defaults__、__code__ 和 __annotations__ 属性，IDE 和框架使用它们提取关于函数签名的信息。但是，为了深入了解这些属性，我们要先探讨 Python 为声明函数形参和传入实参所提供的强大句法。

5.7　从定位参数到仅限关键字参数

Python 最好的特性之一是提供了极为灵活的参数处理机制，而且 Python 3 进一步提供了仅限关键字参数（keyword-only argument）。与之密切相关的是，调用函数时使用 * 和 **“展开”可迭代对象，映射到单个参数。下面通过示例 5-10 中的代码展示这些特性，实际使用的代码在示例 5-11 中。

示例 **5-10**　`tag` 函数用于生成 HTML 标签；使用名为 `cls` 的关键字参数传入“class”属性，这是一种变通方法，因为“ class”是 Python 的关键字

```
def tag(name, *content, cls=None, **attrs):
    """生成一个或多个HTML标签"""
    if cls is not None:
        attrs['class'] = cls
    if attrs:
        attr_str = ''.join(' %s="%s"' % (attr, value)
                           for attr, value
                           in sorted(attrs.items()))
    else:
        attr_str = ''
    if content:
        return '\n'.join('<%s%s>%s</%s>' %
                         (name, attr_str, c, name) for c in content)
    else:
        return '<%s%s />' % (name, attr_str)
```

`tag` 函数的调用方式很多，如示例 5-11 所示。

示例 **5-11**　`tag` 函数（见示例 5-10）众多调用方式中的几种

```
>>> tag('br')  ❶
'<br />'
>>> tag('p', 'hello')  ❷
'<p>hello</p>'
>>> print(tag('p', 'hello', 'world'))
<p>hello</p>
<p>world</p>
```

```
>>> tag('p', 'hello', id=33)  ❸
'<p id="33">hello</p>'
>>> print(tag('p', 'hello', 'world', cls='sidebar'))  ❹
<p class="sidebar">hello</p>
<p class="sidebar">world</p>
>>> tag(content='testing', name="img")  ❺
'<img content="testing" />'
>>> my_tag = {'name': 'img', 'title': 'Sunset Boulevard',
...           'src': 'sunset.jpg', 'cls': 'framed'}
>>> tag(**my_tag)  ❻
'<img class="framed" src="sunset.jpg" title="Sunset Boulevard" />'
```

❶ 传入单个定位参数，生成一个指定名称的空标签。

❷ 第一个参数后面的任意个参数会被 `*content` 捕获，存入一个元组。

❸ `tag` 函数签名中没有明确指定名称的关键字参数会被 `**attrs` 捕获，存入一个字典。

❹ `cls` 参数只能作为关键字参数传入。

❺ 调用 `tag` 函数时，即便第一个定位参数也能作为关键字参数传入。

❻ 在 `my_tag` 前面加上 `**`，字典中的所有元素作为单个参数传入，同名键会绑定到对应的具名参数上，余下的则被 `**attrs` 捕获。

仅限关键字参数是 Python 3 新增的特性。在示例 5-10 中，`cls` 参数只能通过关键字参数指定，它一定不会捕获未命名的定位参数。定义函数时若想指定仅限关键字参数，要把它们放到前面有 `*` 的参数后面。如果不想支持数量不定的定位参数，但是想支持仅限关键字参数，在签名中放一个 `*`，如下所示：

```
>>> def f(a, *, b):
...     return a, b
...
>>> f(1, b=2)
(1, 2)
```

注意，仅限关键字参数不一定要有默认值，可以像上例中 `b` 那样，强制必须传入实参。

下面说明函数参数的内省，以一个 Web 框架中的示例为引子，然后再讨论内省技术。

5.8　获取关于参数的信息

HTTP 微框架 Bobo 中有个使用函数内省的好例子。示例 5-12 是对 Bobo 教程中“Hello world”应用的改编，说明了内省怎么使用。

示例 **5-12**　Bobo 知道 `hello` 需要 `person` 参数，并且从 HTTP 请求中获取它

```
import bobo

@bobo.query('/')
def hello(person):
    return 'Hello %s!' % person
```

`bobo.query` 装饰器把一个普通的函数（如 `hello`）与框架的请求处理机制集成起来了。装饰器会在第 7 章讨论，这不是这个示例的关键。这里的关键是，Bobo 会内省 `hello` 函数，发现它需要一个名为 `person` 的参数，然后从请求中获取那个名称对应的参数，将其传给 `hello` 函数，因此程序员根本不用触碰请求对象。

安装 Bobo，然后启动开发服务器，执行示例 5-12 中的脚本（例如，`bobo -f hello.py`）。访问 `http://localhost:8080/` 看到的消息是“Missing form variable person”，HTTP 状态码是 403。这是因为，Bobo 知道调用 `hello` 函数必须传入 `person` 参数，但是在请求中找不到同名参数。示例 5-13 在 shell 会话中使用 `curl` 展示了这个行为。

示例 **5-13**　如果请求中缺少函数的参数，Bobo 返回 403 forbidden 响应；`curl -i` 的作用是把首部转储到标准输出

```
$ curl -i http://localhost:8080/
HTTP/1.0 403 Forbidden
Date: Thu, 21 Aug 2014 21:39:44 GMT
Server: WSGIServer/0.2 CPython/3.4.1
Content-Type: text/html; charset=UTF-8
Content-Length: 103

<html>
```

```
<head><title>Missing parameter</title></head>
<body>Missing form variable person</body>
</html>
```

然而，如果访问 `http://localhost:8080/?person=Jim`，响应会变成字符串 `'Hello Jim!'`，如示例 5-14 所示。

示例 **5-14**　传入所需的 `person` 参数才能得到 `OK` 响应

```
$ curl -i http://localhost:8080/?person=Jim
HTTP/1.0 200 OK
Date: Thu, 21 Aug 2014 21:42:32 GMT
Server: WSGIServer/0.2 CPython/3.4.1
Content-Type: text/html; charset=UTF-8
Content-Length: 10

Hello Jim!
```

Bobo 是怎么知道函数需要哪个参数的呢？它又是怎么知道参数有没有默认值呢？

函数对象有个 `__defaults__` 属性，它的值是一个元组，里面保存着定位参数和关键字参数的默认值。仅限关键字参数的默认值在 `__kwdefaults__` 属性中。然而，参数的名称在 `__code__` 属性中，它的值是一个 `code` 对象引用，自身也有很多属性。

为了说明这些属性的用途，下面在 clip.py 模块中定义 `clip` 函数，如示例 5-15 所示，然后再审查它。

示例 **5-15**　在指定长度附近截断字符串的函数

```
def clip(text, max_len=80):
    """在max_len前面或后面的第一个空格处截断文本
    """
    end = None
    if len(text) > max_len:
        space_before = text.rfind(' ', 0, max_len)
        if space_before >= 0:
            end = space_before
        else:
```

```
        space_after = text.rfind(' ', max_len)
        if space_after >= 0:
            end = space_after
if end is None:  # 没找到空格
    end = len(text)
return text[:end].rstrip()
```

示例 5-16 审查示例 5-15 中定义的 `clip` 函数，查看 `__defaults__`、`__code__.co_varnames` 和 `__code__.co_argcount` 的值。

示例 **5-16**　提取关于函数参数的信息

```
>>> from clip import clip
>>> clip.__defaults__
(80,)
>>> clip.__code__  # doctest: +ELLIPSIS
<code object clip at 0x...>
>>> clip.__code__.co_varnames
('text', 'max_len', 'end', 'space_before', 'space_after')
>>> clip.__code__.co_argcount
2
```

可以看出，这种组织信息的方式并不是最便利的。参数名称在 `__code__.co_varnames` 中，不过里面还有函数定义体中创建的局部变量。因此，参数名称是前 *N* 个字符串，*N* 的值由 `__code__.co_argcount` 确定。顺便说一下，这里不包含前缀为 `*` 或 `**` 的变长参数。参数的默认值只能通过它们在 `__defaults__` 元组中的位置确定，因此要从后向前扫描才能把参数和默认值对应起来。在这个示例中 `clip` 函数有两个参数，`text` 和 `max_len`，其中一个有默认值，即 `80`，因此它必然属于最后一个参数，即 `max_len`。这有违常理。

幸好，我们有更好的方式——使用 `inspect` 模块。

下面来看一下示例 5-17。

示例 **5-17**　提取函数的签名 [2]

[2]在 Python 3.5 中，本示例的 sig 的值是：<Signature (text, max_len=80)>。——编者注

```
>>> from clip import clip
>>> from inspect import signature
>>> sig = signature(clip)
>>> sig  # doctest: +ELLIPSIS
<inspect.Signature object at 0x...>
>>> str(sig)
'(text, max_len=80)'
>>> for name, param in sig.parameters.items():
...     print(param.kind, ':', name, '=', param.default)
...
POSITIONAL_OR_KEYWORD : text = <class 'inspect._empty'>
POSITIONAL_OR_KEYWORD : max_len = 80
```

这样就好多了。inspect.signature 函数返回一个 inspect.Signature 对象，它有一个 parameters 属性，这是一个有序映射，把参数名和 inspect.Parameter 对象对应起来。各个 Parameter 属性也有自己的属性，例如 name、default 和 kind。特殊的 inspect._empty 值表示没有默认值，考虑到 None 是有效的默认值（也经常这么做），而且这么做是合理的。

kind 属性的值是 _ParameterKind 类中的 5 个值之一，列举如下。

POSITIONAL_OR_KEYWORD

可以通过定位参数和关键字参数传入的形参（多数 Python 函数的参数属于此类）。

VAR_POSITIONAL

定位参数元组。

VAR_KEYWORD

关键字参数字典。

KEYWORD_ONLY

仅限关键字参数（Python 3 新增）。

`POSITIONAL_ONLY`

仅限定位参数；目前，Python 声明函数的句法不支持，但是有些使用 C 语言实现且不接受关键字参数的函数（如 `divmod`）支持。

除了 `name`、`default` 和 `kind`，`inspect.Parameter` 对象还有一个 `annotation`（注解）属性，它的值通常是 `inspect._empty`，但是可能包含 Python 3 新的注解句法提供的函数签名元数据（注解在下一节讨论）。

`inspect.Signature` 对象有个 `bind` 方法，它可以把任意个参数绑定到签名中的形参上，所用的规则与实参到形参的匹配方式一样。框架可以使用这个方法在真正调用函数前验证参数，如示例 5-18 所示。

示例 **5-18**　把`tag` 函数（见示例 5-10）的签名绑定到一个参数字典上 [3]

> [3]在 Python 3.5 中，本示例的 `bound_args` 的值是：`<BoundArguments (name='img', cls='framed', attrs={'title': 'Sunset Boulevard', 'src': 'sunset.jpg'})>`。——编者注

```
>>> import inspect
>>> sig = inspect.signature(tag)  ❶
>>> my_tag = {'name': 'img', 'title': 'Sunset Boulevard',
...           'src': 'sunset.jpg', 'cls': 'framed'}
>>> bound_args = sig.bind(**my_tag)  ❷
>>> bound_args
<inspect.BoundArguments object at 0x...>  ❸
>>> for name, value in bound_args.arguments.items():  ❹
...     print(name, '=', value)
...
name = img
cls = framed
attrs = {'title': 'Sunset Boulevard', 'src': 'sunset.jpg'}
>>> del my_tag['name']  ❺
>>> bound_args = sig.bind(**my_tag)  ❻
Traceback (most recent call last):
  ...
TypeError: 'name' parameter lacking default value
```

❶ 获取 `tag` 函数（见示例 5-10）的签名。

❷ 把一个字典参数传给 .bind() 方法。

❸ 得到一个 inspect.BoundArguments 对象。

❹ 迭代 bound_args.arguments（一个 OrderedDict 对象）中的元素，显示参数的名称和值。

❺ 把必须指定的参数 name 从 my_tag 中删除。

❻ 调用 sig.bind(**my_tag)，抛出 TypeError，抱怨缺少 name 参数。

这个示例在 inspect 模块的帮助下，展示了 Python 数据模型把实参绑定给函数调用中的形参的机制，这与解释器使用的机制相同。

框架和 IDE 等工具可以使用这些信息验证代码。Python 3 的另一个特性——函数注解——增进了这些信息的用途，参见下一节。

5.9 函数注解

Python 3 提供了一种句法，用于为函数声明中的参数和返回值附加元数据。示例 5-19 是示例 5-15 添加注解后的版本，二者唯一的区别在第一行。

示例 **5-19** 有注解的 `clip` 函数

```
def clip(text:str, max_len:'int > 0'=80) -> str:  ❶
    """在max_len前面或后面的第一个空格处截断文本
    """
    end = None
    if len(text) > max_len:
        space_before = text.rfind(' ', 0, max_len)
        if space_before >= 0:
            end = space_before
        else:
            space_after = text.rfind(' ', max_len)
            if space_after >= 0:
                end = space_after
    if end is None:  # 没找到空格
        end = len(text)
    return text[:end].rstrip()
```

❶ 有注解的函数声明。

函数声明中的各个参数可以在 `:` 之后增加注解表达式。如果参数有默认值，注解放在参数名和 `=` 号之间。如果想注解返回值，在 `)` 和函数声明末尾的 `:` 之间添加 `->` 和一个表达式。那个表达式可以是任何类型。注解中最常用的类型是类（如 `str` 或 `int`）和字符串（如 `'int > 0'`）。在示例 5-19 中，`max_len` 参数的注解用的是字符串。

注解不会做任何处理，只是存储在函数的 `__annotations__` 属性（一个字典）中：

```
>>> from clip_annot import clip
>>> clip.__annotations__
{'text': <class 'str'>, 'max_len': 'int > 0', 'return': <class 'str'>}
```

'return' 键保存的是返回值注解，即示例 5-19 中函数声明里以 -> 标记的部分。

Python 对注解所做的唯一的事情是，把它们存储在函数的 __annotations__ 属性里。仅此而已，Python 不做检查、不做强制、不做验证，什么操作都不做。换句话说，注解对 Python 解释器没有任何意义。注解只是元数据，可以供 IDE、框架和装饰器等工具使用。写作本书时，标准库中还没有什么会用到这些元数据，唯有 inspect.signature() 函数知道怎么提取注解，如示例 5-20 所示。

示例 **5-20**　从函数签名中提取注解

```
>>> from clip_annot import clip
>>> from inspect import signature
>>> sig = signature(clip)
>>> sig.return_annotation
<class 'str'>
>>> for param in sig.parameters.values():
...     note = repr(param.annotation).ljust(13)
...     print(note, ':', param.name, '=', param.default)
<class 'str'> : text = <class 'inspect._empty'>
'int > 0'     : max_len = 80
```

signature 函数返回一个 Signature 对象，它有一个 return_annotation 属性和一个 parameters 属性，后者是一个字典，把参数名映射到 Parameter 对象上。每个 Parameter 对象自己也有 annotation 属性。示例 5-20 用到了这几个属性。

在未来，Bobo 等框架可以支持注解，并进一步自动处理请求。例如，使用 price:float 注解的参数可以自动把查询字符串转换成函数期待的 float 类型；quantity:'int > 0' 这样的字符串注解可以转换成对参数的验证。

函数注解的最大影响或许不是让 Bobo 等框架自动设置，而是为 IDE 和 lint 程序等工具中的静态类型检查功能提供额外的类型信息。

深入分析函数之后，本章余下的内容介绍标准库中为函数式编程提供支持的常用包。

5.10　支持函数式编程的包

虽然 Guido 明确表明，Python 的目标不是变成函数式编程语言，但是得益于 `operator` 和 `functools` 等包的支持，函数式编程风格也可以信手拈来。接下来的两节分别介绍这两个包。

5.10.1　operator模块

在函数式编程中，经常需要把算术运算符当作函数使用。例如，不使用递归计算阶乘。求和可以使用 `sum` 函数，但是求积则没有这样的函数。我们可以使用 `reduce` 函数（5.2.1 节是这么做的），但是需要一个函数计算序列中两个元素之积。示例 5-21 展示如何使用 `lambda` 表达式解决这个问题。

示例 **5-21**　使用 `reduce` 函数和一个匿名函数计算阶乘

```
from functools import reduce
def fact(n):
    return reduce(lambda a, b: a*b, range(1, n+1))
```

`operator` 模块为多个算术运算符提供了对应的函数，从而避免编写 `lambda a, b: a*b` 这种平凡的匿名函数。使用算术运算符函数，可以把示例 5-21 改写成示例 5-22 那样。

示例 **5-22**　使用 `reduce` 和 `operator.mul` 函数计算阶乘

```
from functools import reduce
from operator import mul

def fact(n):
    return reduce(mul, range(1, n+1))
```

`operator` 模块中还有一类函数，能替代从序列中取出元素或读取对象属性的 `lambda` 表达式：因此，`itemgetter` 和 `attrgetter` 其实会自

行构建函数。

示例 5-23 展示了 `itemgetter` 的常见用途：根据元组的某个字段给元组列表排序。在这个示例中，按照国家代码（第 2 个字段）的顺序打印各个城市的信息。其实，`itemgetter(1)` 的作用与 `lambda fields: fields[1]` 一样：创建一个接受集合的函数，返回索引位 1 上的元素。

示例 **5-23**　演示使用 `itemgetter` 排序一个元组列表（数据来自示例 2-8）

```
>>> metro_data = [
...     ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)),
...     ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),
...     ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),
...     ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),
...     ('Sao Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
... ]
>>>
>>> from operator import itemgetter
>>> for city in sorted(metro_data, key=itemgetter(1)):
...     print(city)
...
('Sao Paulo', 'BR', 19.649, (-23.547778, -46.635833))
('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889))
('Tokyo', 'JP', 36.933, (35.689722, 139.691667))
('Mexico City', 'MX', 20.142, (19.433333, -99.133333))
('New York-Newark', 'US', 20.104, (40.808611, -74.020386))
```

如果把多个参数传给 `itemgetter`，它构建的函数会返回提取的值构成的元组：

```
>>> cc_name = itemgetter(1, 0)
>>> for city in metro_data:
...     print(cc_name(city))
...
('JP', 'Tokyo')
('IN', 'Delhi NCR')
('MX', 'Mexico City')
('US', 'New York-Newark')
('BR', 'Sao Paulo')
>>>
```

`itemgetter` 使用 `[]` 运算符，因此它不仅支持序列，还支持映射和任

何实现 __getitem__ 方法的类。

attrgetter 与 itemgetter 作用类似，它创建的函数根据名称提取对象的属性。如果把多个属性名传给 attrgetter，它也会返回提取的值构成的元组。此外，如果参数名中包含 .（点号），attrgetter 会深入嵌套对象，获取指定的属性。这些行为如示例 5-24 所示。这个控制台会话不短，因为我们要构建一个嵌套结构，这样才能展示 attrgetter 如何处理包含点号的属性名。

示例 **5-24** 定义一个 namedtuple，名为 metro_data（与示例 5-23 中的列表相同），演示使用 attrgetter 处理它

```
>>> from collections import namedtuple
>>> LatLong = namedtuple('LatLong', 'lat long')  # ❶
>>> Metropolis = namedtuple('Metropolis', 'name cc pop coord')  # ❷
>>> metro_areas = [Metropolis(name, cc, pop, LatLong(lat, long))  # ❸
...     for name, cc, pop, (lat, long) in metro_data]
>>> metro_areas[0]
Metropolis(name='Tokyo', cc='JP', pop=36.933, coord=LatLong(lat=35.689722,
long=139.691667))
>>> metro_areas[0].coord.lat  # ❹
35.689722
>>> from operator import attrgetter
>>> name_lat = attrgetter('name', 'coord.lat')  # ❺
>>>
>>> for city in sorted(metro_areas, key=attrgetter('coord.lat')):  # ❻
...     print(name_lat(city))  # ❼
...
('Sao Paulo', -23.547778)
('Mexico City', 19.433333)
('Delhi NCR', 28.613889)
('Tokyo', 35.689722)
('New York-Newark', 40.808611)
```

❶ 使用 namedtuple 定义 LatLong。

❷ 再定义 Metropolis。

❸ 使用 Metropolis 实例构建 metro_areas 列表；注意，我们使用嵌套的元组拆包提取 (lat, long)，然后使用它们构建 LatLong，作为 Metropolis 的 coord 属性。

❹ 深入 metro_areas[0]，获取它的纬度。

❺ 定义一个 attrgetter，获取 name 属性和嵌套的 coord.lat 属性。

❻ 再次使用 attrgetter，按照纬度排序城市列表。

❼ 使用标号❺中定义的 attrgetter，只显示城市名和纬度。

下面是 operator 模块中定义的部分函数（省略了以 _ 开头的名称，因为它们基本上是实现细节）：[4]

> [4]Python 3.5 中增加了 imatmul 和 matmul。——编者注

```
>>> [name for name in dir(operator) if not name.startswith('_')]
['abs', 'add', 'and_', 'attrgetter', 'concat', 'contains',
'countOf', 'delitem', 'eq', 'floordiv', 'ge', 'getitem', 'gt',
'iadd', 'iand', 'iconcat', 'ifloordiv', 'ilshift', 'imod', 'imul',
'index', 'indexOf', 'inv', 'invert', 'ior', 'ipow', 'irshift',
'is_', 'is_not', 'isub', 'itemgetter', 'itruediv', 'ixor', 'le',
'length_hint', 'lshift', 'lt', 'methodcaller', 'mod', 'mul', 'ne',
'neg', 'not_', 'or_', 'pos', 'pow', 'rshift', 'setitem', 'sub',
'truediv', 'truth', 'xor']
```

这 52 个名称中大部分的作用不言而喻。以 i 开头、后面是另一个运算符的那些名称（如 iadd、iand 等），对应的是增量赋值运算符（如 +=、&= 等）。如果第一个参数是可变的，那么这些运算符函数会就地修改它；否则，作用与不带 i 的函数一样，直接返回运算结果。

在 operator 模块余下的函数中，我们最后介绍一下 methodcaller。它的作用与 attrgetter 和 itemgetter 类似，它会自行创建函数。methodcaller 创建的函数会在对象上调用参数指定的方法，如示例 5-25 所示。

示例 **5-25**　methodcaller 使用示例：第二个测试展示绑定额外参数的方式

```
>>> from operator import methodcaller
>>> s = 'The time has come'
>>> upcase = methodcaller('upper')
>>> upcase(s)
```

```
'THE TIME HAS COME'
>>> hiphenate = methodcaller('replace', ' ', '-')
>>> hiphenate(s)
'The-time-has-come'
```

示例 5-25 中的第一个测试只是为了展示 methodcaller 的用法，如果想把 str.upper 作为函数使用，只需在 str 类上调用，并传入一个字符串参数，如下所示：

```
>>> str.upper(s)
'THE TIME HAS COME'
```

示例 5-25 中的第二个测试表明，methodcaller 还可以冻结某些参数，也就是部分应用（partial application），这与 functools.partial 函数的作用类似。详情参见下一节。

5.10.2 使用functools.partial冻结参数

functools 模块提供了一系列高阶函数，其中最为人熟知的或许是 reduce，我们在 5.2.1 节已经介绍过。余下的函数中，最有用的是 partial 及其变体，partialmethod。

functools.partial 这个高阶函数用于部分应用一个函数。部分应用是指，基于一个函数创建一个新的可调用对象，把原函数的某些参数固定。使用这个函数可以把接受一个或多个参数的函数改编成需要回调的 API，这样参数更少。示例 5-26 做了简单的演示。

示例 **5-26** 使用 partial 把一个两参数函数改编成需要单参数的可调用对象

```
>>> from operator import mul
>>> from functools import partial
>>> triple = partial(mul, 3)  ❶
>>> triple(7)  ❷
21
>>> list(map(triple, range(1, 10)))  ❸
[3, 6, 9, 12, 15, 18, 21, 24, 27]
```

❶ 使用 `mul` 创建 `triple` 函数，把第一个定位参数定为 `3`。

❷ 测试 `triple` 函数。

❸ 在 `map` 中使用 `triple`；在这个示例中不能使用 `mul`。

使用 4.6 节介绍的 `unicode.normalize` 函数再举个例子，这个示例更有实际意义。如果处理多国语言编写的文本，在比较或排序之前可能会想使用 `unicode.normalize('NFC', s)` 处理所有字符串 `s`。如果经常这么做，可以定义一个 `nfc` 函数，如示例 5-27 所示。

示例 **5-27**　使用 `partial` 构建一个便利的 Unicode 规范化函数

```
>>> import unicodedata, functools
>>> nfc = functools.partial(unicodedata.normalize, 'NFC')
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> s1, s2
('café', 'café')
>>> s1 == s2
False
>>> nfc(s1) == nfc(s2)
True
```

`partial` 的第一个参数是一个可调用对象，后面跟着任意个要绑定的定位参数和关键字参数。

示例 5-28 在示例 5-10 中定义的 `tag` 函数上使用 `partial`，冻结一个定位参数和一个关键字参数。

示例 **5-28**　把 `partial` 应用到示例 5-10 中定义的 `tag` 函数上

```
>>> from tagger import tag
>>> tag
<function tag at 0x10206d1e0>  ❶
>>> from functools import partial
>>> picture = partial(tag, 'img', cls='pic-frame')  ❷
>>> picture(src='wumpus.jpeg')
'<img class="pic-frame" src="wumpus.jpeg" />'  ❸
>>> picture
functools.partial(<function tag at 0x10206d1e0>, 'img', cls='pic-frame')  ❹
```

```
>>> picture.func  ❺
<function tag at 0x10206d1e0>
>>> picture.args
('img',)
>>> picture.keywords
{'cls': 'pic-frame'}
```

❶ 从示例 5-10 中导入 `tag` 函数，查看它的 ID。

❷ 使用 `tag` 创建 `picture` 函数，把第一个定位参数固定为 `'img'`，把 `cls` 关键字参数固定为 `'pic-frame'`。

❸ `picture` 的行为符合预期。

❹ `partial()` 返回一个 `functools.partial` 对象。[5]

> [5]functools.py 的源码（https://hg.python.org/cpython/file/default/Lib/functools.py）表明，`functools.partial` 类是使用 C 语言实现的，而且默认使用这个实现。如果这个实现不可用，从 Python 3.4 起，`functools` 模块为 `partial` 提供了纯 Python 实现。

❺ `functools.partial` 对象提供了访问原函数和固定参数的属性。

`functools.partialmethod` 函数（Python 3.4 新增）的作用与 `partial` 一样，不过是用于处理方法的。

`functools` 模块中的 `lru_cache` 函数令人印象深刻，它会做备忘（memoization），这是一种自动优化措施，它会存储耗时的函数调用结果，避免重新计算。第 7 章将会介绍这个函数，还将讨论装饰器，以及旨在用作装饰器的其他高阶函数：`singledispatch` 和 `wraps`。

5.11　本章小结

本章的目标是探讨 Python 函数的一等本性。这意味着，我们可以把函数赋值给变量、传给其他函数、存储在数据结构中，以及访问函数的属性，供框架和一些工具使用。高阶函数是函数式编程的重要组成部分，即使现在不像以前那样经常使用 `map`、`filter` 和 `reduce` 函数了，但是还有列表推导（以及类似的结构，如生成器表达式）以及 `sum`、`all` 和 `any` 等内置的归约函数。Python 中常用的高阶函数有内置函数 `sorted`、`min`、`max` 和 `functools. partial`。

Python 有 7 种可调用对象，从 `lambda` 表达式创建的简单函数到实现 `__call__` 方法的类实例。这些可调用对象都能通过内置的 `callable()` 函数检测。每一种可调用对象都支持使用相同的丰富句法声明形式参数，包括仅限关键字参数和注解——二者都是 Python 3 引入的新特性。

Python 函数及其注解有丰富的属性，在 `inspect` 模块的帮助下，可以读取它们。例如，`Signature.bind` 方法使用灵活的规则把实参绑定到形参上，这与 Python 使用的规则一样。

最后，本章介绍了 `operator` 模块中的一些函数，以及 `functools.partial` 函数，有了这些函数，函数式编程就不太需要功能有限的 `lambda` 表达式了。

5.12　延伸阅读

接下来的两章继续探讨使用函数对象编程。第 6 章说明一等函数如何简化某些经典的面向对象设计模式，第 7 章说明函数装饰器（一种特别的高阶函数）和支持装饰器的闭包机制。

《Python Cookbook（第 3 版）中文版》（David Beazley 和 Brian K. Jones 著）的第 7 章是对本章和第 7 章很好的补充，那一章基本上使用不同的方式探讨了相同的概念。

Python 语言参考手册中的“3.2. The standard type hierarchy”一节（https://docs.python.org/3/reference/datamodel.html#the-standard-type-hierarchy）对 7 种可调用类型和其他所有内置类型做了介绍。

本章讨论的 Python 3 专有特性有各自的 PEP：“PEP 3102—Keyword-Only Arguments”（https://www.python.org/dev/peps/pep-3102/）和“PEP 3107—Function Annotations”（https://www.python.org/dev/peps/pep-3107/）。

若想进一步了解目前对注解的使用，Stack Overflow 网站中有两个问答值得一读：一个是“What are good uses for Python3's‘Function Annotations’”（http://stackoverflow.com/questions/3038033/what-are-good-uses-for-python3s-function-annotations），Raymond Hettinger 给出了务实的回答和深入的见解；另一个是“What good are Python function annotations?”（http://stackoverflow.com/questions/13784713/what-good-are-python-function-annotations），某 个回答中大量引用了 Guido van Rossum 的观点。

如果你想使用 `inspect` 模块，“PEP 362—Function Signature Object”（https://www.python.org/dev/peps/pep-0362/）值得一读，可以帮助了解实现细节。

A. M. Kuchling 的文章“Python Functional Programming HOWTO”（http://docs.python.org/3/howto/functional.html）对 Python 函数式编程做了很好的介绍。不过，那篇文章的重点是使用迭代器和生成器，这是第 14 章的话题。

`fn.py`（https://github.com/kachayev/fn.py）是为 Python 2 和 Python 3 提供函数式编程支持的包。据作者 Alexey Kachayev 介绍，`fn.py` 提供了 Python“所缺少的函数式特性”。这个包提供的 `@recur.tco` 装饰器为 Python 中的无限递归实现了尾调用优化。此外，`fn.py` 还提供了很多其他函数、数据结构和诀窍。

Stack Overflow 网站中的问题“Python: Why is functools.partial necessary?”（http://stackoverflow.com/questions/3252228/python-why-is-functools-partial-necessary）有个详实（而有趣）的回答，答主是 Alex Martelli，他是经典的《Python 技术手册》一书的作者。

Jim Fulton 开发的 Bobo 或许是第一个称得上是面向对象的 Web 框架。如果你对这个框架感兴趣，想进一步学习它最近的重写版本，先从“Introduction”（http://bobo.readthedocs.io/en/latest/）入手。在 Joel Spolsky 的博客中，Phillip J. Eby 在评论中提到了 Bobo 的一些早期历史（http://discuss.fogcreek.com/joelonsoftware/default.asp?cmd=show&ixPost=94006）。

杂谈

关于Bobo

我的 Python 编程生涯从 Bobo 开始。1998 年，我在自己的第一个 Python Web 项目中使用了 Bobo。当时我在寻找编写 Web 应用的面向对象方式，尝试过一些 Perl 和 Java框架之后，我发现了 Bobo。

1997 年，Bobo 开创了对象发布概念：直接把 URL 映射到对象层次结构上，无需配置路由。看到这种做法的精妙之处后，我被 Bobo 吸引住了。Bobo 还能通过分析处理请求的方法或函数的签名来自动处理 HTTP 查询。

Bobo 由 Jim Fulton 创建，他被人称为“Zope 教皇”（The Zope Pope），因为他在 Zope 框架的开发中的起到领衔作用。Zope 是 Plone CMS、SchoolTool、ERP5 和其他大型 Python 项目的基础。Jim 还是 ZODB（Zope Object Database）的创建者，这是一个事务型对象数据库，提供了 ACID（“atomicity, consistency, isolation, and durability”，原子性、一致性、隔离性和耐久性），它的设计目的是简化 Python 的使用。

后来，为了支持 WSGI 和现代的 Python 版本（包括 Python 3），Jim 从头重写了 Bobo。写作本书时，Bobo 使用 `six` 库做函数内省，这是为了兼容 Python 2 和 Python 3，因为这两个版本在函数对象和相关的 API 上做了修改。

Python 是函数式语言吗

2000 年左右，我在美国做培训，Guido van Rossum 到访了教室（他不是讲师）。在课后的问答环节，有人问他 Python 的哪些特性是从其他语言借鉴而来的。他答道："Python 中一切好的特性都是从其他语言中借鉴来的。"

布朗大学的计算机科学教授 Shriram Krishnamurthi 在其论文"Teaching Programming Languages in a Post-Linnaean Age"（http://cs.brown.edu/~sk/Publications/Papers/Published/sk-teach-pl-post-linnaean/）的开头这样写道：

> 编程语言"范式"已近末日，它们是旧时代的遗留物，令人厌烦。既然现代语言的设计者对范式不屑一顾，那么我们的课程为什么要像奴隶一样对其言听计从？

在那篇论文中，下面这一段点名提到了 Python：

> 对 Python、Ruby 或 Perl 这些语言还要了解什么呢？它们的设计者没有耐心去精确实现林奈层次结构；设计者按照自己的意愿从别处借鉴特性，创建出完全无视过往概念的大杂烩。

Krishnamurthi 指出，不要试图把语言归为某一类；相反，把它们视作特性的聚合更有用。

为 Python 提供一等函数打开了函数式编程的大门，不过这并不是 Guido 的目的。他在"Origins of Python's Functional Features"一文（http://python-history.blogspot.com/2009/04/origins-of-pythons-functional-features.html）中说，`map`、`filter` 和 `reduce` 的最初目的是为 Python 增加 `lambda` 表达式。这些特性都由 Amrit Prem 贡献，添加在 1994 年发布的 Python 1.0 中（参见 CPython 源码中的 Misc/HISTORY 文件，https://hg.python.org/cpython/file/default/Misc/HISTORY）。

`lambda`、`map`、`filter` 和 `reduce` 首次出现在 Lisp 中，这是最早的一门函数式语言。然而，Lisp 没有限制在 lambda 表达式中能做什么，因为 Lisp 中的一切都是表达式。 Python 使用的是面向语句的句法，表达式中不能包含语句，而很多语言结构都是语句，例如 `try/catch`，我编写 `lambda` 表达式时最想念这个语句。Python 为了提高句法的可读性，必须付出这样的代价。[6]Lisp 有很多优点，可读性一定不是其中之一。

讽刺的是，从另一门函数式语言（Haskell）中借用列表推导之后，Python 对 `map`、`filter`，以及 `lambda` 表达式的需求极大地减少了。

除了匿名函数句法上的限制之外，影响函数式编程惯用法在 Python 中广泛使用的最大障碍是缺少尾递归消除（tail-recursion elimination），这是一项优化措施，在函数的定义体“末尾”递归调用，从而提高计算函数的内存使用效率。Guido 在另一篇博客文章（“Tail Recursion Elimination”，http://neopythonic.blogspot.com/2009/04/tail-recursion-elimination.html）中解释了为什么这种优化措施不适合 Python。这篇文章详细讨论了技术论证，不过前三个也是最重要的原因与易用性有关。Python 作为一门易于使用、学习和教授的语言并非偶然，有 Guido 在为我们把关。

综上，从设计上看，不管函数式语言的定义如何，Python 都不是一门函数式语言。Python 只是从函数式语言中借鉴了一些好的想法。

匿名函数的问题

除了 Python 独有的句法上的局限，在任何一门语言中，匿名函数都有一个严重的缺点：没有名称。

我是半开玩笑的。函数有名称，栈跟踪更易于阅读。匿名函数是一种便利的简洁方式，人们乐于使用它们，但是有时会忘乎所以，尤其是在鼓励深层嵌套匿名函数的语言和环境中，如 JavaScript 和 Node.js。匿名函数嵌套的层级太深，不利于调试和处理错误。Python 中的异步编程结构更好，或许就是因为 `lambda` 表达式有局限。我保证，后面会进一步讨论异步编程，但是必须等到第 18 章。顺便说一下，promise 对象、期物（future）和 deferred 对象是

现代异步 API 中使用的概念。把它们与协程结合起来，能避免掉入“回调地狱”。18.5 节会说明如何不用回调来做异步编程。

> [6]此外，还有一个问题：把代码粘贴到 Web 论坛时，缩进会丢失。当然，这是题外话。

第 6 章　使用一等函数实现设计模式

> 符合模式并不表示做得对。[1]
>
> ——Ralph Johnson
> 经典的《设计模式：可复用面向对象软件的基础》的作者之一

[1]出自 2014 年 11 月 15 日 Ralph Johnson 在圣保罗大学 IME/CCSL 所做的演讲，“Root Cause Analysis of Some Faults in Design Patterns”。

虽然设计模式与语言无关，但这并不意味着每一个模式都能在每一门语言中使用。1996 年，Peter Norvig 在题为“Design Patterns in Dynamic Languages”（http://norvig.com/design-patterns/）的演讲中指出，Gamma 等人合著的《设计模式：可复用面向对象软件的基础》一书中有 23 个模式，其中有 16 个在动态语言中“不见了，或者简化了”（参见第 9 张幻灯片）。他讨论的是 Lisp 和 Dylan，不过很多相关的动态特性在 Python 中也能找到。

《设计模式：可复用面向对象软件的基础》的作者在引言中承认，所用的语言决定了哪些模式可用：

> 程序设计语言的选择非常重要，它将影响人们理解问题的出发点。我们的设计模式采用了 Smalltalk 和 C++ 层的语言特性，这个选择实际上决定了哪些机制可以方便地实现，而哪些则不能。若我们采用过程式语言，可能就要包括诸如“集成”“封装”和“多态”的设计模式。相应地，一些特殊的面向对象语言可以直接支持我们的某些模式，例如 CLOS 支持多方法概念，这就减少了访问者模式的必要性。[2]

[2]《设计模式：可复用面向对象软件的基础》第 3 页。

具体而言，Norvig 建议在有一等函数的语言中重新审视“策略”“命令”“模板方法”和“访问者”模式。通常，我们可以把这些模式中涉及的

某些类的实例替换成简单的函数，从而减少样板代码。本章将使用函数对象重构“策略”模式，还将讨论一种更简单的方式，用于简化“命令”模式。

6.1　案例分析：重构“策略”模式

如果合理利用作为一等对象的函数，某些设计模式可以简化，“策略”模式就是其中一个很好的例子。本节接下来的内容中将说明“策略”模式，并使用《设计模式：可复用面向对象软件的基础》一书中所述的“经典”结构实现它。如果你熟悉这个经典模式，可以跳到 6.1.2 节，了解如何使用函数重构代码来有效减少代码行数。

6.1.1　经典的“策略”模式

图 6-1 中的 UML 类图指出了“策略”模式对类的编排。

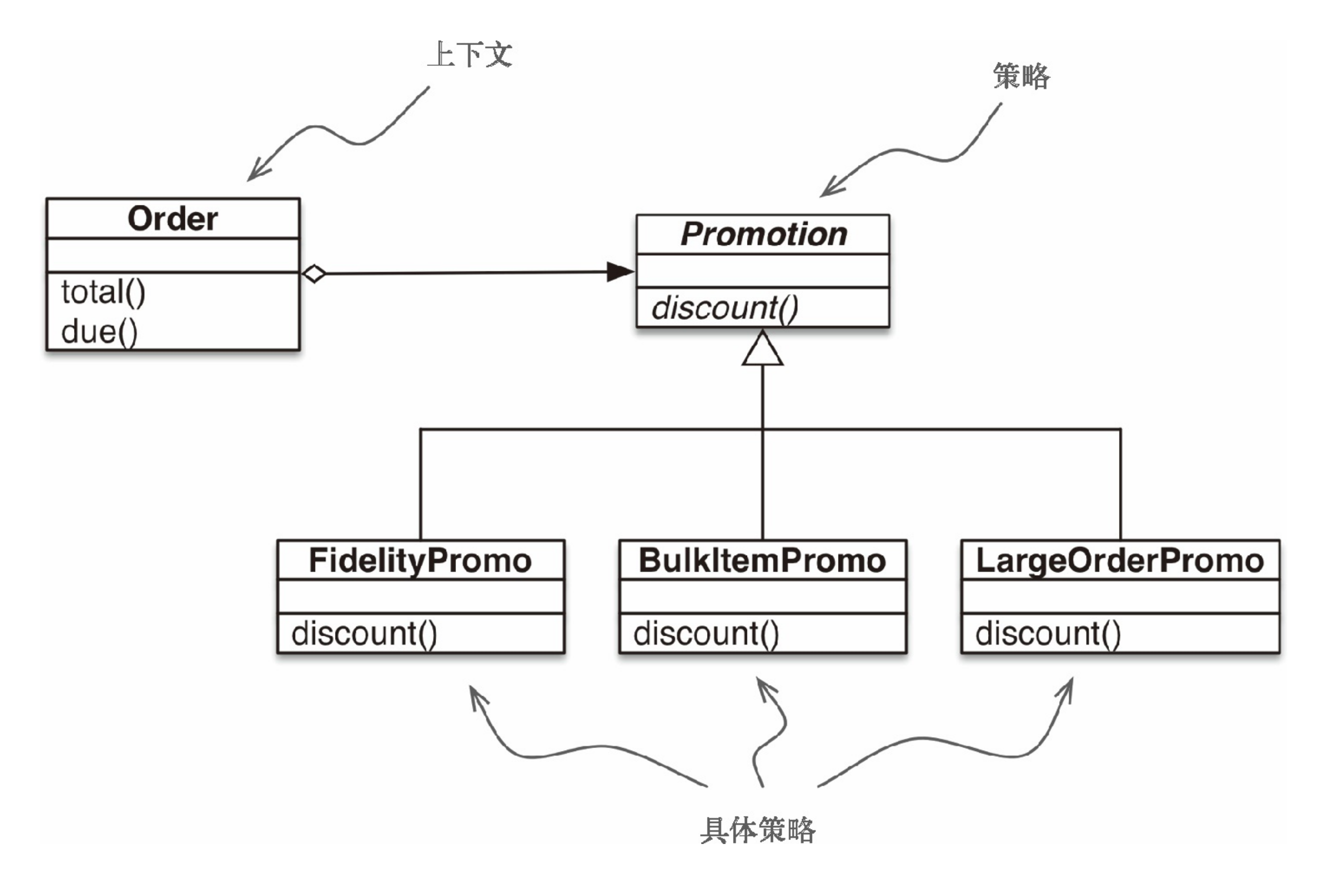


图 **6-1**：使用“策略”设计模式处理订单折扣的 **UML** 类图

《设计模式：可复用面向对象软件的基础》一书是这样概述“策略”模式的：

> 定义一系列算法，把它们一一封装起来，并且使它们可以相互替换。本模式使得算法可以独立于使用它的客户而变化。

电商领域有个功能明显可以使用“策略”模式，即根据客户的属性或订单中的商品计算折扣。

假如一个网店制定了下述折扣规则。

- 有 1000 或以上积分的顾客，每个订单享 5% 折扣。
- 同一订单中，单个商品的数量达到 20 个或以上，享 10% 折扣。
- 订单中的不同商品达到 10 个或以上，享 7% 折扣。

简单起见，我们假定一个订单一次只能享用一个折扣。

“策略”模式的 UML 类图见图 6-1，其中涉及下列内容。

上下文

　　把一些计算委托给实现不同算法的可互换组件，它提供服务。在这个电商示例中，上下文是 `Order`，它会根据不同的算法计算促销折扣。

策略

　　实现不同算法的组件共同的接口。在这个示例中，名为 `Promotion` 的抽象类扮演这个角色。

具体策略

　　“策略”的具体子类。`fidelityPromo`、`BulkPromo` 和 `LargeOrderPromo` 是这里实现的三个具体策略。

示例 6-1 实现了图 6-1 中的方案。按照《设计模式：可复用面向对象软件的基础》一书的说明，具体策略由上下文类的客户选择。在这个示例中，实例化订单之前，系统会以某种方式选择一种促销折扣策略，然后把它传给 `Order` 构造方法。具体怎么选择策略，不在这个模式的职责范围内。

示例 **6-1**　实现 `Order` 类，支持插入式折扣策略

```
from abc import ABC, abstractmethod
from collections import namedtuple

Customer = namedtuple('Customer', 'name fidelity')


class LineItem:

    def __init__(self, product, quantity, price):
        self.product = product
        self.quantity = quantity
        self.price = price

    def total(self):
        return self.price * self.quantity
```

```
class Order:  # 上下文

    def __init__(self, customer, cart, promotion=None):
        self.customer = customer
        self.cart = list(cart)
        self.promotion = promotion

    def total(self):
        if not hasattr(self, '__total'):
            self.__total = sum(item.total() for item in self.cart)
        return self.__total

    def due(self):
        if self.promotion is None:
            discount = 0
        else:
            discount = self.promotion.discount(self)
        return self.total() - discount

    def __repr__(self):
        fmt = '<Order total: {:.2f} due: {:.2f}>'
        return fmt.format(self.total(), self.due())


class Promotion(ABC) : # 策略：抽象基类

    @abstractmethod
    def discount(self, order):
        """返回折扣金额（正值）"""


class FidelityPromo(Promotion):  # 第一个具体策略
    """为积分为1000或以上的顾客提供5%折扣"""

    def discount(self, order):
        return order.total() * .05 if order.customer.fidelity >= 1000 else 0


class BulkItemPromo(Promotion):  # 第二个具体策略
    """单个商品为20个或以上时提供10%折扣"""

    def discount(self, order):
        discount = 0
        for item in order.cart:
            if item.quantity >= 20:
                discount += item.total() * .1
        return discount
```

```
class LargeOrderPromo(Promotion):  # 第三个具体策略
    """订单中的不同商品达到10个或以上时提供7%折扣"""

    def discount(self, order):
        distinct_items = {item.product for item in order.cart}
        if len(distinct_items) >= 10:
            return order.total() * .07
        return 0
```

注意，在示例 6-1 中，我把 `Promotion` 定义为抽象基类（Abstract Base Class，ABC），这么做是为了使用 `@abstractmethod` 装饰器，从而明确表明所用的模式。

> 在 Python 3.4 中，声明抽象基类最简单的方式是子类化 `abc.ABC`。我在示例 6-1 中就是这么做的。从 Python 3.0 到 Python 3.3，必须在 `class` 语句中使用 `metaclass=` 关键字（例如，`class Promotion(metaclass=ABCMeta):`）。

示例 6-2 是一些 doctest，在某个实现了上述规则的模块中演示和验证相关操作。

示例 **6-2**　使用不同促销折扣的 `Order` 类示例

```
    >>> joe = Customer('John Doe', 0) ❶
    >>> ann = Customer('Ann Smith', 1100)
    >>> cart = [LineItem('banana', 4, .5), ❷
    ...         LineItem('apple', 10, 1.5),
    ...         LineItem('watermellon', 5, 5.0)]
    >>> Order(joe, cart, FidelityPromo()) ❸
    <Order total: 42.00 due: 42.00>
    >>> Order(ann, cart, FidelityPromo()) ❹
    <Order total: 42.00 due: 39.90>
    >>> banana_cart = [LineItem('banana', 30, .5), ❺
    ...                LineItem('apple', 10, 1.5)]
    >>> Order(joe, banana_cart, BulkItemPromo()) ❻
    <Order total: 30.00 due: 28.50>
    >>> long_order = [LineItem(str(item_code), 1, 1.0) ❼
    ...               for item_code in range(10)]
    >>> Order(joe, long_order, LargeOrderPromo()) ❽
```

```
    <Order total: 10.00 due: 9.30>
    >>> Order(joe, cart, LargeOrderPromo())
    <Order total: 42.00 due: 42.00>
```

❶ 两个顾客：`joe` 的积分是 0，`ann` 的积分是 1100。

❷ 有三个商品的购物车。

❸ `fidelityPromo` 没给 `joe` 提供折扣。

❹ `ann` 得到了 5% 折扣，因为她的积分超过 1000。

❺ `banana_cart` 中有 30 把香蕉和 10 个苹果。

❻ `BulkItemPromo` 为 `joe` 购买的香蕉优惠了 1.50 美元。

❼ `long_order` 中有 10 个不同的商品，每个商品的价格为 1.00 美元。

❽ `LargerOrderPromo` 为 `joe` 的整个订单提供了 7% 折扣。

示例 6-1 完全可用，但是利用 Python 中作为对象的函数，可以使用更少的代码实现相同的功能。详情参见下一节。

6.1.2　使用函数实现“策略”模式

在示例 6-1 中，每个具体策略都是一个类，而且都只定义了一个方法，即 `discount`。此外，策略实例没有状态（没有实例属性）。你可能会说，它们看起来像是普通的函数——的确如此。示例 6-3 是对示例 6-1 的重构，把具体策略换成了简单的函数，而且去掉了 `Promo` 抽象类。

示例 **6-3**　`Order` 类和使用函数实现的折扣策略

```
from collections import namedtuple

Customer = namedtuple('Customer', 'name fidelity')


class LineItem:

    def __init__(self, product, quantity, price):
```

```
        self.product = product
        self.quantity = quantity
        self.price = price

    def total(self):
        return self.price * self.quantity


class Order:  # 上下文

    def __init__(self, customer, cart, promotion=None):
        self.customer = customer
        self.cart = list(cart)
        self.promotion = promotion

    def total(self):
        if not hasattr(self, '__total'):
            self.__total = sum(item.total() for item in self.cart)
        return self.__total

    def due(self):
        if self.promotion is None:
            discount = 0
        else:
            discount = self.promotion(self)  ❶
        return self.total() - discount

    def __repr__(self):
        fmt = '<Order total: {:.2f} due: {:.2f}>'
        return fmt.format(self.total(), self.due())

❷

def fidelity_promo(order):  ❸
    """为积分为1000或以上的顾客提供5%折扣"""
    return order.total() * .05 if order.customer.fidelity >= 1000 else 0


def bulk_item_promo(order):
    """单个商品为20个或以上时提供10%折扣"""
    discount = 0
    for item in order.cart:
        if item.quantity >= 20:
            discount += item.total() * .1
    return discount

def large_order_promo(order):
```

```
    """订单中的不同商品达到10个或以上时提供7%折扣"""
    distinct_items = {item.product for item in order.cart}
    if len(distinct_items) >= 10:
        return order.total() * .07
    return 0
```

❶ 计算折扣只需调用 `self.promotion()` 函数。

❷ 没有抽象类。

❸ 各个策略都是函数。

示例 6-3 中的代码比示例 6-1 少 12 行。不仅如此，新的 `Order` 类使用起来更简单，如示例 6-4 中的 doctest 所示。

示例 **6-4**　使用函数实现的促销折扣的 `Order` 类示例

```
    >>> joe = Customer('John Doe', 0)  ❶
    >>> ann = Customer('Ann Smith', 1100)
    >>> cart = [LineItem('banana', 4, .5),
    ...         LineItem('apple', 10, 1.5),
    ...         LineItem('watermellon', 5, 5.0)]
    >>> Order(joe, cart, fidelity_promo)  ❷
    <Order total: 42.00 due: 42.00>
    >>> Order(ann, cart, fidelity_promo)
    <Order total: 42.00 due: 39.90>
    >>> banana_cart = [LineItem('banana', 30, .5),
    ...                LineItem('apple', 10, 1.5)]
    >>> Order(joe, banana_cart, bulk_item_promo)  ❸
    <Order total: 30.00 due: 28.50>
    >>> long_order = [LineItem(str(item_code), 1, 1.0)
    ...               for item_code in range(10)]
    >>> Order(joe, long_order, large_order_promo)
    <Order total: 10.00 due: 9.30>
    >>> Order(joe, cart, large_order_promo)
    <Order total: 42.00 due: 42.00>
```

❶ 与示例 6-1 一样的测试固件。

❷ 为了把折扣策略应用到 `Order` 实例上，只需把促销函数作为参数传入。

❸ 这个测试和下一个测试使用不同的促销函数。

注意示例 6-4 中的标注：没必要在新建订单时实例化新的促销对象，函数拿来即用。

值得注意的是，《设计模式：可复用面向对象软件的基础》一书的作者指出："策略对象通常是很好的享元（flyweight）。"[3] 那本书的另一部分对"享元"下了定义："享元是可共享的对象，可以同时在多个上下文中使用。"[4] 共享是推荐的做法，这样不必在每个新的上下文（这里是 `Order` 实例）中使用相同的策略时不断新建具体策略对象，从而减少消耗。因此，为了避免"策略"模式的一个缺点（运行时消耗），《设计模式：可复用面向对象软件的基础》的作者建议再使用另一个模式。但此时，代码行数和维护成本会不断攀升。

[3]《设计模式：可复用面向对象软件的基础》第 214 页。

[4]《设计模式：可复用面向对象软件的基础》第 129 页。

在复杂的情况下，需要具体策略维护内部状态时，可能需要把"策略"和"享元"模式结合起来。但是，具体策略一般没有内部状态，只是处理上下文中的数据。此时，一定要使用普通的函数，别去编写只有一个方法的类，再去实现另一个类声明的单函数接口。函数比用户定义的类的实例轻量，而且无需使用"享元"模式，因为各个策略函数在 Python 编译模块时只会创建一次。普通的函数也是"可共享的对象，可以同时在多个上下文中使用"。

至此，我们使用函数实现了"策略"模式，由此也出现了其他可能性。假设我们想创建一个"元策略"，让它为指定的订单选择最佳折扣。接下来的几节会接着重构，利用函数和模块是对象，使用不同的方式实现这个需求。

6.1.3　选择最佳策略：简单的方式

我们继续使用示例 6-4 中的顾客和购物车，在此基础上添加 3 个测试，如示例 6-5 所示。

示例 **6-5**　`best_promo` 函数计算所有折扣，并返回额度最大的

```
>>> Order(joe, long_order, best_promo)  ❶
<Order total: 10.00 due: 9.30>
>>> Order(joe, banana_cart, best_promo)  ❷
<Order total: 30.00 due: 28.50>
>>> Order(ann, cart, best_promo)  ❸
<Order total: 42.00 due: 39.90>
```

❶ best_promo 为顾客 joe 选择 larger_order_promo。

❷ 订购大量香蕉时，joe 使用 bulk_item_promo 提供的折扣。

❸ 在一个简单的购物车中，best_promo 为忠实顾客 ann 提供 fidelity_promo 优惠的折扣。

best_promo 函数的实现特别简单，如示例 6-6 所示。

示例 **6-6**　best_promo 迭代一个函数列表，并找出折扣额度最大的

```
promos = [fidelity_promo, bulk_item_promo, large_order_promo]  ❶

def best_promo(order):  ❷
    """选择可用的最佳折扣
    """
    return max(promo(order) for promo in promos)  ❸
```

❶ promos 列出以函数实现的各个策略。

❷ 与其他几个 *_promo 函数一样，best_promo 函数的参数是一个 Order 实例。

❸ 使用生成器表达式把 order 传给 promos 列表中的各个函数，返回折扣额度最大的那个函数。

示例 6-6 简单明了，promos 是函数列表。习惯函数是一等对象后，自然而然就会构建那种数据结构存储函数。

虽然示例 6-6 可用，而且易于阅读，但是有些重复可能会导致不易察觉的缺陷：若想添加新的促销策略，要定义相应的函数，还要记得把它添

加到 promos 列表中；否则，当新促销函数显式地作为参数传给 Order 时，它是可用的，但是 best_promo 不会考虑它。

继续往下读，了解这个问题的几种解决方案。

6.1.4 找出模块中的全部策略

在 Python 中，模块也是一等对象，而且标准库提供了几个处理模块的函数。Python 文档是这样说明内置函数 globals 的。

globals()

返回一个字典，表示当前的全局符号表。这个符号表始终针对当前模块（对函数或方法来说，是指定义它们的模块，而不是调用它们的模块）。

示例 6-7 使用 globals 函数帮助 best_promo 自动找到其他可用的 *_promo 函数，过程有点曲折。

示例 **6-7** 内省模块的全局命名空间，构建 promos 列表

```
promos = [globals()[name] for name in globals()  ❶
            if name.endswith('_promo')  ❷
            and name != 'best_promo']   ❸

def best_promo(order):
    """选择可用的最佳折扣
    """
    return max(promo(order) for promo in promos)  ❹
```

❶ 迭代 globals() 返回字典中的各个 name。

❷ 只选择以 _promo 结尾的名称。

❸ 过滤掉 best_promo 自身，防止无限递归。

❹ best_promo 内部的代码没有变化。

收集所有可用促销的另一种方法是，在一个单独的模块中保存所有策略函数，把 best_promo 排除在外。

在示例 6-8 中，最大的变化是内省名为 promotions 的独立模块，构建策略函数列表。注意，示例 6-8 要导入 promotions 模块，以及提供高阶内省函数的 inspect 模块（简单起见，这里没有给出导入语句，因为导入语句一般放在文件顶部）。

示例 **6-8** 内省单独的 promotions 模块，构建 promos 列表

```
promos = [func for name, func in
                inspect.getmembers(promotions, inspect.isfunction)]

def best_promo(order):
    """选择可用的最佳折扣
    """
    return max(promo(order) for promo in promos)
```

inspect.getmembers 函数用于获取对象（这里是 promotions 模块）的属性，第二个参数是可选的判断条件（一个布尔值函数）。我们使用的是 inspect.isfunction，只获取模块中的函数。

不管怎么命名策略函数，示例 6-8 都可用；唯一重要的是，promotions 模块只能包含计算订单折扣的函数。当然，这是对代码的隐性假设。如果有人在 promotions 模块中使用不同的签名定义函数，那么 best_promo 函数尝试将其应用到订单上时会出错。

我们可以添加更为严格的测试，审查传给实例的参数，进一步过滤函数。示例 6-8 的目的不是提供完善的方案，而是强调模块内省的一种用途。

动态收集促销折扣函数更为显式的一种方案是使用简单的装饰器。第 7 章讨论函数装饰器时会使用其他方式实现这个电商“策略”模式示例。

下一节讨论“命令”模式。这个设计模式也常使用单方法类实现，同样也可以换成普通的函数。

6.2 “命令”模式

“命令”设计模式也可以通过把函数作为参数传递而简化。这一模式对类的编排如图 6-2 所示。

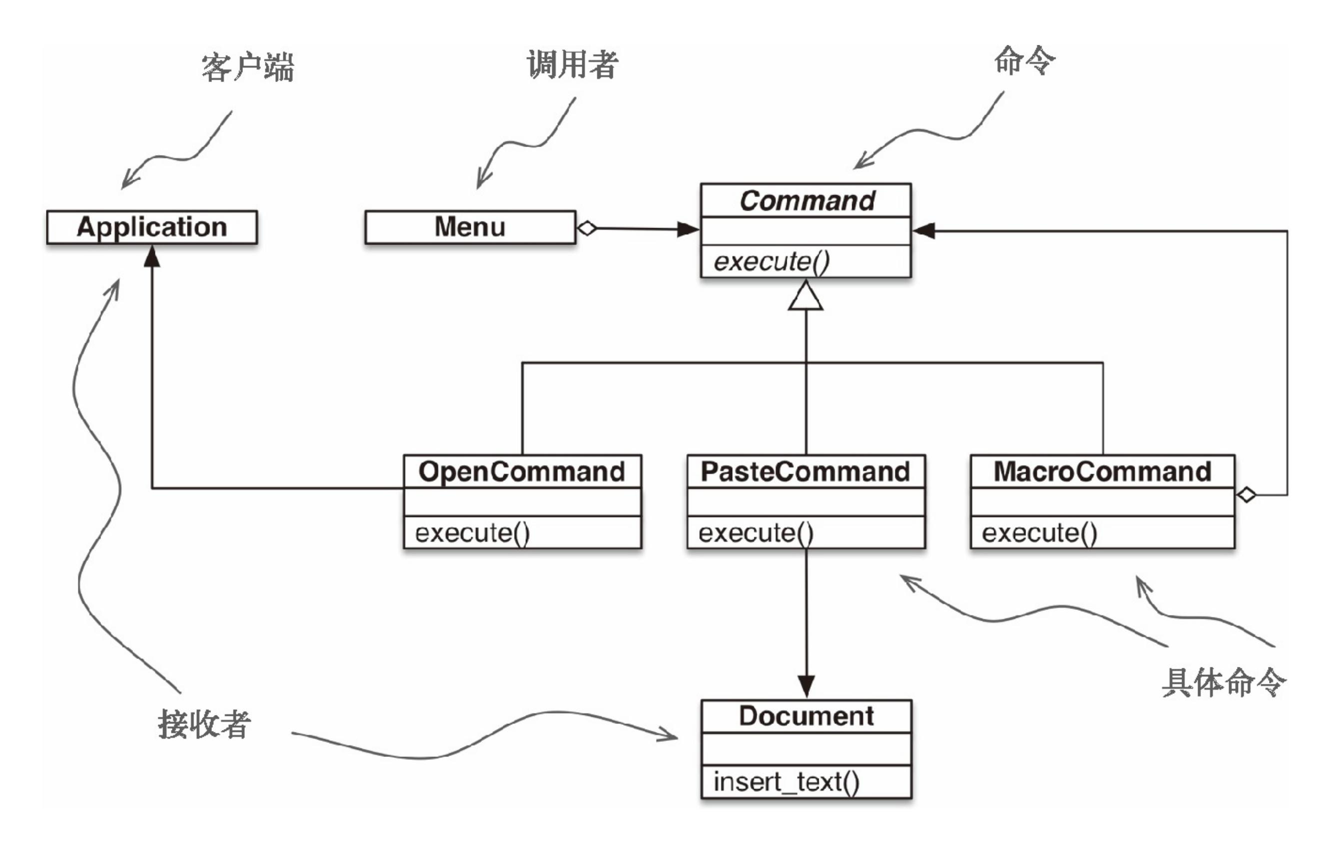


图 **6-2**：菜单驱动的文本编辑器的 **UML** 类图，使用“命令”设计模式实现。各个命令可以有不同的接收者（实现操作的对象）。对 **`PasteCommand`** 来说，接收者是 **`Document`**。对 **`OpenCommand`** 来说，接收者是应用程序

“命令”模式的目的是解耦调用操作的对象（调用者）和提供实现的对象（接收者）。在《设计模式：可复用面向对象软件的基础》所举的示例中，调用者是图形应用程序中的菜单项，而接收者是被编辑的文档或应用程序自身。

这个模式的做法是，在二者之间放一个 `Command` 对象，让它实现只有一个方法（`execute`）的接口，调用接收者中的方法执行所需的操作。这样，调用者无需了解接收者的接口，而且不同的接收者可以适应不同

的 Command 子类。调用者有一个具体的命令，通过调用 execute 方法执行。注意，图 6-2 中的 MacroCommand 可能保存一系列命令，它的 execute() 方法会在各个命令上调用相同的方法。

Gamma 等人说过："命令模式是回调机制的面向对象替代品。"问题是，我们需要回调机制的面向对象替代品吗？有时确实需要，但并非始终需要。

我们可以不为调用者提供一个 Command 实例，而是给它一个函数。此时，调用者不用调用 command.execute()，直接调用 command() 即可。MacroCommand 可以实现成定义了 __call__ 方法的类。这样，MacroCommand 的实例就是可调用对象，各自维护着一个函数列表，供以后调用，如示例 6-9 所示。

示例 **6-9** MacroCommand 的各个实例都在内部存储着命令列表

```
class MacroCommand:
    """一个执行一组命令的命令"""

    def __init__(self, commands):
        self.commands = list(commands) # ❶

    def __call__(self):
        for command in self.commands: # ❷
            command()
```

❶ 使用 commands 参数构建一个列表，这样能确保参数是可迭代对象，还能在各个 MacroCommand 实例中保存各个命令引用的副本。

❷ 调用 MacroCommand 实例时，self.commands 中的各个命令依序执行。

复杂的"命令"模式（如支持撤销操作）可能需要更多，而不仅是简单的回调函数。即便如此，也可以考虑使用 Python 提供的几个替代品。

- 像示例 6-9 中 MacroCommand 那样的可调用实例，可以保存任何所需的状态，而且除了 __call__ 之外还可以提供其他方法。

- 可以使用闭包在调用之间保存函数的内部状态。

使用一等函数对“命令”模式的重新审视到此结束。站在一定高度上看，这里采用的方式与“策略”模式所用的类似：把实现单方法接口的类的实例替换成可调用对象。毕竟，每个 Python 可调用对象都实现了单方法接口，这个方法就是 `__call__`。

6.3　本章小结

经典的《设计模式：可复用面向对象软件的基础》一书出版几年后，Peter Norvig 指出，“在 Lisp 或 Dylan 中，23 个设计模式中有 16 个的实现方式比 C++ 中更简单，而且能保持同等质量，至少各个模式的某些用途如此”（Norvig 的“Design Patterns in Dynamic Languages”演讲，第 9 张幻灯片，http://www.norvig.com/design-patterns/index.htm）。Python 有些动态特性与 Lisp 和 Dylan 一样，尤其是本书这一部分着重讨论的一等函数。

本章开头引用的那句话是 Ralph Johnson 在纪念《设计模式：可复用面向对象软件的基础》原书出版 20 周年的活动上所说的，他指出这本书的缺点之一是：“过多强调设计模式的结果，而没有细说过程。”[5] 本章从“策略”模式开始，使用一等函数简化了实现方式。

> [5]与本章开头引用的那句话同出一处：2014 年 11 月 15 日 Johnson 在 IME-USP 所做的演讲，“Root Cause Analysis of Some Faults in Design Patterns”。

很多情况下，在 Python 中使用函数或可调用对象实现回调更自然，这比模仿 Gamma、Helm、Johnson 和 Vlissides 在书中所述的“策略”或“命令”模式要好。本章对“策略”模式的重构和对“命令”模式的讨论是为了通过示例说明一个更为常见的做法：有时，设计模式或 API 要求组件实现单方法接口，而那个方法的名称很宽泛，例如“execute”“run”或“doIt”。在 Python 中，这些模式或 API 通常可以使用一等函数或其他可调用的对象实现，从而减少样板代码。

Peter Norvig 那次设计模式演讲想表达的观点是，“命令”和“策略”模式（以及“模板方法”和“访问者”模式）可以使用一等函数实现，这样更简单，甚至“不见了”，至少对这些模式的某些用途来说是如此。

6.4 延伸阅读

结束对“策略”模式的讨论时，我建议使用函数装饰器改进示例 6-8。本章还多次提到了闭包。装饰器和闭包是第 7 章的话题。那一章首先重构本章的电商示例，使用装饰器注册可用的促销方式。

《Python Cookbook（第 3 版）中文版》（David Beazley 和 Brian K. Jones 著）的“8.21 实现访问者模式”使用优雅的方式实现了“访问者”模式，其中的 `NodeVisitor` 类把方法当作一等对象处理。

在设计模式方面，Python 程序员的阅读选择没有其他语言多。

据我所知，截至 2014 年 6 月，*Learning Python Design Patterns*（Gennadiy Zlobin 著，Packt 出版社）是唯一一本专门针对 Python 设计模式的书。不过 Zlobin 这本书特别薄（100 页），只涵盖了 23 种设计模式中的 8 种。

《Python 高级编程》（Tarek Ziadé著）是市面上最好的 Python 中级书，第 14 章“有用的设计模式”从 Python 程序员的视角介绍了 7 种经典模式。

Alex Martelli 做过几次关于 Python 设计模式的演讲。他在 EuroPython 2011 上的演讲有视频（http://pyvideo.org/europython-2011/python-design-patterns.html），他的个人网站中有一些幻灯片（http://www.aleax.it/gdd_pydp.pdf）。这些年，我找到了不同的幻灯片和视频，长短不一，因此要仔细搜索他的名字和“Python Design Patterns”这些词。

2008 年左右，《Java 编程思想》的作者 Bruce Eckel 开始写一本题为 *Python 3 Patterns, Recipes and Idioms* 的书（http://www.mindviewinc.com/Books/Python3Patterns/Index.php）。这本书有很多贡献者，领头人是 Eckel，但是六年过去了，依然没有写完，看样子是流产了（写作本书时，仓库的最后一次改动是在两年前 [6]）。

[6]至本书中文版出版时，仓库的最后一次改动是在 2015 年 8 月 4 日。——编者注

用 Java 写的设计模式书很多，其中我最喜欢的一本是《Head First 设计模式》（Eric Freeman、Bert Bates、Kathy Sierra 和 Elisabeth Robson 著）。这本书讲解了 23 个经典模式中的 16 个。如果你喜欢 Head First 系列丛书的古怪风格，而且想了解这个主题，你会喜欢这本书的。不过，它是围绕 Java 讲解的。

如果想换个新鲜的角度，从支持鸭子类型和一等函数的动态语言入手，《Ruby 设计模式》（Russ Olsen 著）一书有很多见解也适用于 Python。虽然 Python 和 Ruby 在句法上有很多区别，但是二者在语义方面很接近，比 Java 或 C++ 接近。

在“Design Patterns in Dynamic Languages”（http://norvig.com/design-patterns/）这一演讲中，Peter Norvig 展示了如何使用一等函数（和其他动态特性）简化几个经典的设计模式，或者根本不需要使用设计模式。

当然，如果你想认真研究这个话题，Gamma 等人写的《设计模式：可复用面向对象软件的基础》一书是必读的。光是“引言”就值回书钱了。人们经常引用这本书中的两个设计原则：“对接口编程，而不是对实现编程”和“优先使用对象组合，而不是类继承”。

杂谈

Python 拥有一等函数和一等类型，Norvig 声称，这些特性对 23 个模式中的 16 个有影响（“Design Patterns in Dynamic Languages”，第 10 张幻灯片，http://norvig.com/design-patterns/）。读到下一章你会发现，Python 还有泛函数（7.8.2 节）。泛函数与 CLOS 中的多方法（multimethod）类似，Gamma 等人建议使用多方法以一种简单的方式实现经典的“访问者”模式。Norvig 却说，多方法能简化“生成器”（Builder）模式（第 10 张幻灯片）。可见，设计模式与语言特性无法精确对应。

世界各地的课堂经常使用 Java 示例讲解设计模式。我不止一次听学生说过，他们以为设计模式在任何语言中都有用。事实证明，在 Gamma 等人合著的那本书中，尽管大部分使用 C++ 代码说明（少数使用 Smalltalk），但是 23 个“经典的”设计模式都能很好地在“经典的”Java 中运用。然而，这并不意味着所有模式都能一成不变地在任何语言中运用。那本书的作者在开头就明确表明了，“一些特殊的面向对象语言可以直接支持我们的某些模式”（完整的引用见

本章开头）。

与 Java、C++ 或 Ruby 相比，Python 设计模式方面的书籍都很薄。延伸阅读中提到的 *Learning Python Design Patterns*（Gennadiy Zlobin 著）在 2013 年 11 月才出版。而《Ruby 设计模式》（Russ Olsen 著）在 2007 年就出版了，而且有 384 页，比 Zlobin 的那本书多出 284 页。

如今，Python 在学术界越来越流行，希望以后会有更多以这门语言讲解设计模式的书籍。此外，Java 8 引入了方法引用和匿名函数，这些广受期盼的特性有可能为 Java 催生新的模式实现方式——要知道，语言会进化，因此运用经典设计模式的方式必定要随之进化。

第 7 章 函数装饰器和闭包

> 有很多人抱怨，把这个特性命名为“装饰器”不好。主要原因是，这个名称与 GoF 书[1] 使用的不一致。装饰器这个名称可能更适合在编译器领域使用，因为它会遍历并注解句法树。
>
> ——“PEP 318 — Decorators for Functions and Methods”

> [1]指 1995 年出版的英文原版《设计模式：可复用面向对象软件的基础》，作者是四个人，人们称之为“四人组”（Gang of Four）。

函数装饰器用于在源码中“标记”函数，以某种方式增强函数的行为。这是一项强大的功能，但是若想掌握，必须理解闭包。

`nonlocal` 是新近出现的保留关键字，在 Python 3.0 中引入。作为 Python 程序员，如果严格遵守基于类的面向对象编程方式，即便不知道这个关键字也不会受到影响。然而，如果你想自己实现函数装饰器，那就必须了解闭包的方方面面，因此也就需要知道 `nonlocal`。

除了在装饰器中有用处之外，闭包还是回调式异步编程和函数式编程风格的基础。

本章的最终目标是解释清楚函数装饰器的工作原理，包括最简单的注册装饰器和较复杂的参数化装饰器。但是，在实现这一目标之前，我们要讨论下述话题：

- Python 如何计算装饰器句法
- Python 如何判断变量是不是局部的
- 闭包存在的原因和工作原理
- `nonlocal` 能解决什么问题

掌握这些基础知识后，我们可以进一步探讨装饰器：

- 实现行为良好的装饰器
- 标准库中有用的装饰器
- 实现一个参数化装饰器

下面将首先介绍装饰器的基础知识，然后再讨论上面列出的各个话题。

7.1　装饰器基础知识

装饰器是可调用的对象，其参数是另一个函数（被装饰的函数）。[2] 装饰器可能会处理被装饰的函数，然后把它返回，或者将其替换成另一个函数或可调用对象。

> [2]Python 也支持类装饰器，参见第 21 章。

假如有个名为 decorate 的装饰器：

```
@decorate
def target():
    print('running target()')
```

上述代码的效果与下述写法一样：

```
def target():
    print('running target()')

target = decorate(target)
```

两种写法的最终结果一样：上述两个代码片段执行完毕后得到的 target 不一定是原来那个 target 函数，而是 decorate(target) 返回的函数。

为了确认被装饰的函数会被替换，请看示例 7-1 中的控制台会话。

示例 **7-1**　装饰器通常把函数替换成另一个函数

```
>>> def deco(func):
...     def inner():
...         print('running inner()')
...     return inner  ❶
...
>>> @deco
... def target():  ❷
```

```
...     print('running target()')
...
>>> target()  ❸
running inner()
>>> target  ❹
<function deco.<locals>.inner at 0x10063b598>
```

❶ deco 返回 inner 函数对象。

❷ 使用 deco 装饰 target。

❸ 调用被装饰的 target 其实会运行 inner。

❹ 审查对象，发现 target 现在是 inner 的引用。

严格来说，装饰器只是语法糖。如前所示，装饰器可以像常规的可调用对象那样调用，其参数是另一个函数。有时，这样做更方便，尤其是做元编程（在运行时改变程序的行为）时。

综上，装饰器的一大特性是，能把被装饰的函数替换成其他函数。第二个特性是，装饰器在加载模块时立即执行。下一节会说明。

7.2 Python何时执行装饰器

装饰器的一个关键特性是，它们在被装饰的函数定义之后立即运行。这通常是在导入时（即 Python 加载模块时），如示例 7-2 中的 registration.py 模块所示。

示例 **7-2** registration.py 模块

```
registry = []  ❶

def register(func):  ❷
    print('running register(%s)' % func)  ❸
    registry.append(func)  ❹
    return func  ❺

@register  ❻
def f1():
    print('running f1()')

@register
def f2():
    print('running f2()')

def f3():  ❼
    print('running f3()')

def main():  ❽
    print('running main()')
    print('registry ->', registry)
    f1()
    f2()
    f3()

if __name__=='__main__':
    main()  ❾
```

❶ `registry` 保存被 `@register` 装饰的函数引用。

❷ `register` 的参数是一个函数。

❸ 为了演示，显示被装饰的函数。

❹ 把 `func` 存入 `registry`。

❺ 返回 `func`：必须返回函数；这里返回的函数与通过参数传入的一样。

❻ `f1` 和 `f2` 被 `@register` 装饰。

❼ `f3` 没有装饰。

❽ `main` 显示 `registry`，然后调用 `f1()`、`f2()` 和 `f3()`。

❾ 只有把 registration.py 当作脚本运行时才调用 `main()`。

把 registration.py 当作脚本运行得到的输出如下：

```
$ python3 registration.py
running register(<function f1 at 0x100631bf8>)
running register(<function f2 at 0x100631c80>)
running main()
registry -> [<function f1 at 0x100631bf8>, <function f2 at 0x100631c80>]
running f1()
running f2()
running f3()
```

注意，`register` 在模块中其他函数之前运行（两次）。调用 `register` 时，传给它的参数是被装饰的函数，例如 `<function f1 at 0x100631bf8>`。

加载模块后，`registry` 中有两个被装饰函数的引用：`f1` 和 `f2`。这两个函数，以及 `f3`，只在 `main` 明确调用它们时才执行。

如果导入 registration.py 模块（不作为脚本运行），输出如下：

```
>>> import registration
running register(<function f1 at 0x10063b1e0>)
running register(<function f2 at 0x10063b268>)
```

此时查看 `registry` 的值，得到的输出如下：

```
>>> registration.registry
[<function f1 at 0x10063b1e0>, <function f2 at 0x10063b268>]
```

示例 7-2 主要想强调，函数装饰器在导入模块时立即执行，而被装饰的函数只在明确调用时运行。这突出了 Python 程序员所说的导入时和运行时之间的区别。

考虑到装饰器在真实代码中的常用方式，示例 7-2 有两个不寻常的地方。

- 装饰器函数与被装饰的函数在同一个模块中定义。实际情况是，装饰器通常在一个模块中定义，然后应用到其他模块中的函数上。

- `register` 装饰器返回的函数与通过参数传入的相同。实际上，大多数装饰器会在内部定义一个函数，然后将其返回。

虽然示例 7-2 中的 `register` 装饰器原封不动地返回被装饰的函数，但是这种技术并非没有用处。很多 Python Web 框架使用这样的装饰器把函数添加到某种中央注册处，例如把 URL 模式映射到生成 HTTP 响应的函数上的注册处。这种注册装饰器可能会也可能不会修改被装饰的函数。下一节会举例说明。

7.3　使用装饰器改进“策略”模式

使用注册装饰器可以改进 6.1 节中的电商促销折扣示例。

回顾一下，示例 6-6 的主要问题是，定义体中有函数的名称，但是 best_promo 用来判断哪个折扣幅度最大的 promos 列表中也有函数名称。这种重复是个问题，因为新增策略函数后可能会忘记把它添加到 promos 列表中，导致 best_promo 忽略新策略，而且不报错，为系统引入了不易察觉的缺陷。示例 7-3 使用注册装饰器解决了这个问题。

示例 **7-3**　promos 列表中的值使用 promotion 装饰器填充

```
promos = []  ❶

def promotion(promo_func):  ❷
    promos.append(promo_func)
    return promo_func

@promotion  ❸
def fidelity(order):
    """为积分为1000或以上的顾客提供5%折扣"""
    return order.total() * .05 if order.customer.fidelity >= 1000 else 0

@promotion
def bulk_item(order):
    """单个商品为20个或以上时提供10%折扣"""
    discount = 0
    for item in order.cart:
        if item.quantity >= 20:
            discount += item.total() * .1
    return discount

@promotion
def large_order(order):
    """订单中的不同商品达到10个或以上时提供7%折扣"""
    distinct_items = {item.product for item in order.cart}
    if len(distinct_items) >= 10:
        return order.total() * .07
    return 0

def best_promo(order):  ❹
    """选择可用的最佳折扣
```

```
    """
    return max(promo(order) for promo in promos)
```

❶ promos 列表起初是空的。

❷ promotion 把 promo_func 添加到 promos 列表中，然后原封不动地将其返回。

❸ 被 @promotion 装饰的函数都会添加到 promos 列表中。

❹ best_promos 无需修改，因为它依赖 promos 列表。

与 6.1 节给出的方案相比，这个方案有几个优点。

- 促销策略函数无需使用特殊的名称（即不用以 _promo 结尾）。
- @promotion 装饰器突出了被装饰的函数的作用，还便于临时禁用某个促销策略：只需把装饰器注释掉。
- 促销折扣策略可以在其他模块中定义，在系统中的任何地方都行，只要使用 @promotion 装饰即可。

不过，多数装饰器会修改被装饰的函数。通常，它们会定义一个内部函数，然后将其返回，替换被装饰的函数。使用内部函数的代码几乎都要靠闭包才能正确运作。为了理解闭包，我们要退后一步，先了解 Python 中的变量作用域。

7.4　变量作用域规则

在示例 7-4 中，我们定义并测试了一个函数，它读取两个变量的值：一个是局部变量 `a`，是函数的参数；另一个是变量 `b`，这个函数没有定义它。

示例 **7-4**　一个函数，读取一个局部变量和一个全局变量

```
>>> def f1(a):
...     print(a)
...     print(b)
...
>>> f1(3)
3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in f1
NameError: global name 'b' is not defined
```

出现错误并不奇怪。[3] 在示例 7-4 中，如果先给全局变量 `b` 赋值，然后再调用 `f`，那就不会出错：

> [3]在 Python 3.5 中，错误信息是 `NameError: name 'b' is not defined`，删除了 `global`。——编者注

```
>>> b = 6
>>> f1(3)
3
6
```

下面看一个可能会让你吃惊的示例。

看一下示例 7-5 中的 `f2` 函数。前两行代码与示例 7-4 中的 `f1` 一样，然后为 `b` 赋值，再打印它的值。可是，在赋值之前，第二个 `print` 失败了。

示例 **7-5**　`b` 是局部变量，因为在函数的定义体中给它赋值了

```
>>> b = 6
>>> def f2(a):
...     print(a)
...     print(b)
...     b = 9
...
>>> f2(3)
3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in f2
UnboundLocalError: local variable 'b' referenced before assignment
```

注意，首先输出了 3，这表明 print(a) 语句执行了。但是第二个语句 print(b) 执行不了。一开始我很吃惊，我觉得会打印 6，因为有个全局变量 b，而且是在 print(b) 之后为局部变量 b 赋值的。

可事实是，Python 编译函数的定义体时，它判断 b 是局部变量，因为在函数中给它赋值了。生成的字节码证实了这种判断，Python 会尝试从本地环境获取 b。后面调用 f2(3) 时， f2 的定义体会获取并打印局部变量 a 的值，但是尝试获取局部变量 b 的值时，发现 b 没有绑定值。

这不是缺陷，而是设计选择：Python 不要求声明变量，但是假定在函数定义体中赋值的变量是局部变量。这比 JavaScript 的行为好多了，JavaScript 也不要求声明变量，但是如果忘记把变量声明为局部变量（使用 var），可能会在不知情的情况下获取全局变量。

如果在函数中赋值时想让解释器把 b 当成全局变量，要使用 global 声明：

```
>>> b = 6
>>> def f3(a):
...     global b
...     print(a)
...     print(b)
...     b = 9
...

>>> f3(3)
3
6
>>> b
```

```
9

>>> f3(3)
3
9
>>> b = 30
>>> b
30
>>>
```

了解 Python 的变量作用域之后，下一节可以讨论闭包了。如果好奇示例 7-4 和示例 7-5 中的两个函数生成的字节码有什么区别，请阅读下述附注栏。

比较字节码

`dis` 模块为反汇编 Python 函数字节码提供了简单的方式。示例 7-6 和 7-7 中分别是示例 7-4 中 `f1` 和示例 7-5 中 `f2` 的字节码。

示例 **7-6** 反汇编示例 7-4 中的 `f1` 函数

```
>>> from dis import dis
>>> dis(f1)
 2           0 LOAD_GLOBAL              0 (print)  ❶
             3 LOAD_FAST                0 (a)  ❷
             6 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             9 POP_TOP

 3          10 LOAD_GLOBAL              0 (print)
            13 LOAD_GLOBAL              1 (b)  ❸
            16 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
            19 POP_TOP
            20 LOAD_CONST               0 (None)
            23 RETURN_VALUE
```

❶ 加载全局名称 `print`。

❷ 加载本地名称 `a`。

❸ 加载全局名称 `b`。

请比较示例 7-6 中 `f1` 的字节码和示例 7-7 中 `f2` 的字节码。

示例 **7-7**　反汇编示例 7-5 中的 `f2` 函数

```
>>> dis(f2)
 2           0 LOAD_GLOBAL              0 (print)
             3 LOAD_FAST                0 (a)
             6 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             9 POP_TOP

 3          10 LOAD_GLOBAL              0 (print)
            13 LOAD_FAST                1 (b)  ❶
            16 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
            19 POP_TOP

 4          20 LOAD_CONST               1 (9)
            23 STORE_FAST               1 (b)
            26 LOAD_CONST               0 (None)
            29 RETURN_VALUE
```

❶ 加载本地名称 `b`。这表明，编译器把 `b` 视作局部变量，即使在后面才为 `b` 赋值，因为变量的种类（是不是局部变量）不能改变函数的定义体。

运行字节码的 CPython VM 是栈机器，因此 `LOAD` 和 `POP` 操作引用的是栈。深入说明 Python 操作码不在本书范畴之内，不过 `dis` 模块的文档（http://docs.python.org/3/library/dis.html）对其做了说明。

7.5　闭包

在博客圈，人们有时会把闭包和匿名函数弄混。这是有历史原因的：在函数内部定义函数不常见，直到开始使用匿名函数才会这样做。而且，只有涉及嵌套函数时才有闭包问题。因此，很多人是同时知道这两个概念的。

其实，闭包指延伸了作用域的函数，其中包含函数定义体中引用、但是不在定义体中定义的非全局变量。函数是不是匿名的没有关系，关键是它能访问定义体之外定义的非全局变量。

这个概念难以掌握，最好通过示例理解。

假如有个名为 avg 的函数，它的作用是计算不断增加的系列值的均值；例如，整个历史中某个商品的平均收盘价。每天都会增加新价格，因此平均值要考虑至目前为止所有的价格。

起初，avg 是这样使用的：

```
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

avg 从何而来，它又在哪里保存历史值呢？

初学者可能会像示例 7-8 那样使用类实现。

示例 **7-8**　average_oo.py：计算移动平均值的类

```
class Averager():

    def __init__(self):
        self.series = []

    def __call__(self, new_value):
```

```
        self.series.append(new_value)
        total = sum(self.series)
        return total/len(self.series)
```

Averager 的实例是可调用对象：

```
>>> avg = Averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

示例 7-9 是函数式实现，使用高阶函数 make_averager。

示例 **7-9**　average.py：计算移动平均值的高阶函数

```
def make_averager():
    series = []

    def averager(new_value):
        series.append(new_value)
        total = sum(series)
        return total/len(series)

    return averager
```

调用 make_averager 时，返回一个 averager 函数对象。每次调用 averager 时，它会把参数添加到系列值中，然后计算当前平均值，如示例 7-10 所示。

示例 **7-10**　测试示例 7-9

```
>>> avg = make_averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
```

```
11.0
```

注意，这两个示例有共通之处：调用 Averager() 或 make_averager() 得到一个可调用对象 avg，它会更新历史值，然后计算当前均值。在示例 7-8 中，avg 是 Averager 的实例；在示例 7-9 中是内部函数 averager。不管怎样，我们都只需调用 avg(n)，把 n 放入系列值中，然后重新计算均值。

Averager 类的实例 avg 在哪里存储历史值很明显：self.series 实例属性。但是第二个示例中的 avg 函数在哪里寻找 series 呢？

注意，series 是 make_averager 函数的局部变量，因为那个函数的定义体中初始化了 series：series = []。可是，调用 avg(10) 时，make_averager 函数已经返回了，而它的本地作用域也一去不复返了。

在 averager 函数中，series 是自由变量（free variable）。这是一个技术术语，指未在本地作用域中绑定的变量，参见图 7-1。

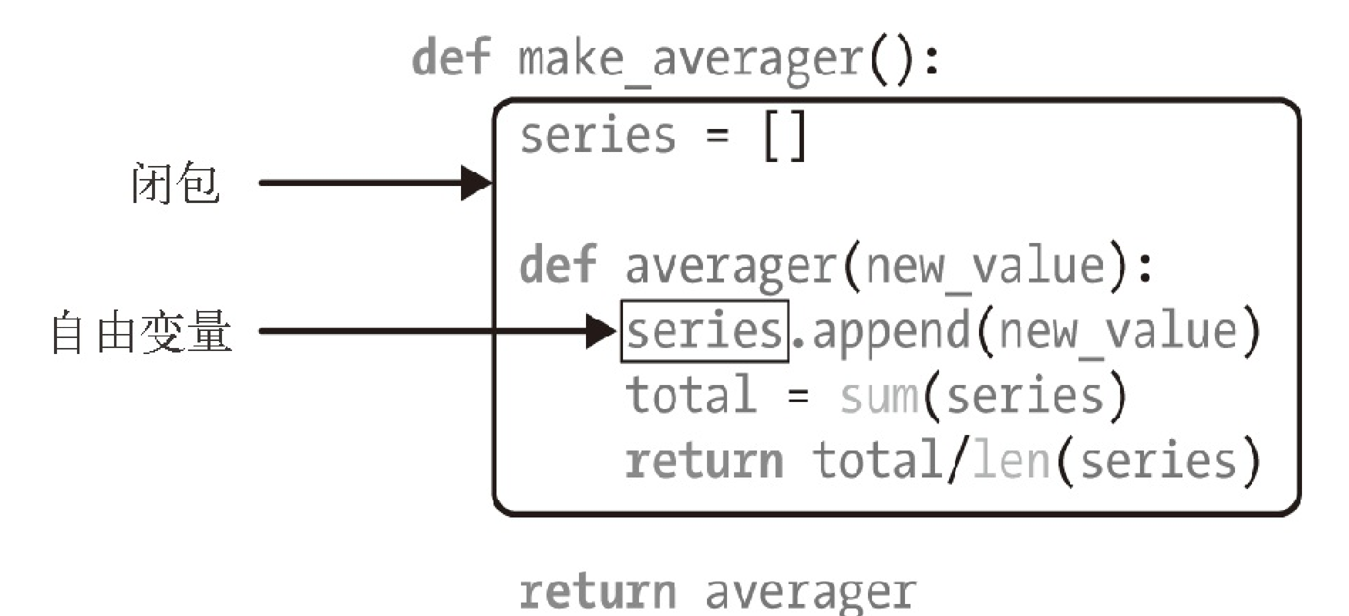


图 **7-1**：**averager** 的闭包延伸到那个函数的作用域之外，包含自由变量 **series** 的绑定

审查返回的 averager 对象，我们发现 Python 在 __code__ 属性（表示编译后的函数定义体）中保存局部变量和自由变量的名称，如示例 7-11

所示。

示例 **7-11**　审查 make_averager（见示例 7-9）创建的函数

```
>>> avg.__code__.co_varnames
('new_value', 'total')
>>> avg.__code__.co_freevars
('series',)
```

series 的绑定在返回的 avg 函数的 __closure__ 属性中。avg.__closure__ 中的各个元素对应于 avg.__code__.co_freevars 中的一个名称。这些元素是 cell 对象，有个 cell_contents 属性，保存着真正的值。这些属性的值如示例 7-12 所示。

示例 **7-12**　接续示例 7-11

```
>>> avg.__code__.co_freevars
('series',)
>>> avg.__closure__
(<cell at 0x107a44f78: list object at 0x107a91a48>,)
>>> avg.__closure__[0].cell_contents
[10, 11, 12]
```

综上，闭包是一种函数，它会保留定义函数时存在的自由变量的绑定，这样调用函数时，虽然定义作用域不可用了，但是仍能使用那些绑定。

注意，只有嵌套在其他函数中的函数才可能需要处理不在全局作用域中的外部变量。

7.6　nonlocal声明

前面实现 make_averager 函数的方法效率不高。在示例 7-9 中，我们把所有值存储在历史数列中，然后在每次调用 averager 时使用 sum 求和。更好的实现方式是，只存储目前的总值和元素个数，然后使用这两个数计算均值。

示例 7-13 中的实现有缺陷，只是为了阐明观点。你能看出缺陷在哪儿吗？

示例 **7-13**　计算移动平均值的高阶函数，不保存所有历史值，但有缺陷

```
def make_averager():
    count = 0
    total = 0

    def averager(new_value):
        count += 1
        total += new_value
        return total / count

    return averager
```

尝试使用示例 7-13 中定义的函数，会得到如下结果：

```
>>> avg = make_averager()
>>> avg(10)
Traceback (most recent call last):
  ...
UnboundLocalError: local variable 'count' referenced before assignment
>>>
```

问题是，当 count 是数字或任何不可变类型时，count += 1 语句的作用其实与 count = count + 1 一样。因此，我们在 averager 的定义体中为 count 赋值了，这会把 count 变成局部变量。total 变量也受这个问题影响。

示例 7-9 没遇到这个问题，因为我们没有给 `series` 赋值，我们只是调用 `series.append`，并把它传给 `sum` 和 `len`。也就是说，我们利用了列表是可变的对象这一事实。

但是对数字、字符串、元组等不可变类型来说，只能读取，不能更新。如果尝试重新绑定，例如 `count = count + 1`，其实会隐式创建局部变量 `count`。这样，`count` 就不是自由变量了，因此不会保存在闭包中。

为了解决这个问题，Python 3 引入了 `nonlocal` 声明。它的作用是把变量标记为自由变量，即使在函数中为变量赋予新值了，也会变成自由变量。如果为 `nonlocal` 声明的变量赋予新值，闭包中保存的绑定会更新。最新版 `make_averager` 的正确实现如示例 7-14 所示。

示例 **7-14**　计算移动平均值，不保存所有历史（使用 `nonlocal` 修正）

```
def make_averager():
    count = 0
    total = 0

    def averager(new_value):
        nonlocal count, total
        count += 1
        total += new_value
        return total / count

    return averager
```



对付没有 **nonlocal** 的 **Python 2**

Python 2 没有 `nonlocal`，因此需要变通方法，“PEP 3104—Access to Names in Outer Scopes”（`nonlocal` 在这个 PEP 中引入，http://www.python.org/dev/peps/pep-3104/）中的第三个代码片段给出了一种方法。基本上，这种处理方式是把内部函数需要修改的变量（如 `count` 和 `total`）存储为可变对象（如字典或简单的实例）的元素或属性，并且把那个对象绑定给一个自由变量。

至此，我们了解了 Python 闭包，下面可以使用嵌套函数正式实现装饰器了。

7.7　实现一个简单的装饰器

示例 7-15 定义了一个装饰器，它会在每次调用被装饰的函数时计时，然后把经过的时间、传入的参数和调用的结果打印出来。

示例 **7-15**　一个简单的装饰器，输出函数的运行时间

```
import time

def clock(func):
    def clocked(*args):  # ❶
        t0 = time.perf_counter()
        result = func(*args)  # ❷
        elapsed = time.perf_counter() - t0
        name = func.__name__
        arg_str = ', '.join(repr(arg) for arg in args)
        print('[%0.8fs] %s(%s) -> %r' % (elapsed, name, arg_str, result))
        return result
    return clocked #  ❸
```

❶ 定义内部函数 `clocked`，它接受任意个定位参数。

❷ 这行代码可用，是因为 `clocked` 的闭包中包含自由变量 `func`。

❸ 返回内部函数，取代被装饰的函数。示例 7-16 演示了 `clock` 装饰器的用法。

示例 **7-16**　使用 `clock` 装饰器

```
# clockdeco_demo.py

import time
from clockdeco import clock

@clock
def snooze(seconds):
    time.sleep(seconds)

@clock
```

```
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)

if __name__=='__main__':
    print('*' * 40, 'Calling snooze(.123)')
    snooze(.123)
    print('*' * 40, 'Calling factorial(6)')
    print('6! =', factorial(6))
```

运行示例 7-16 得到的输出如下：

```
$ python3 clockdeco_demo.py
**************************************** Calling snooze(123)
[0.12405610s] snooze(.123) -> None
**************************************** Calling factorial(6)
[0.00000191s] factorial(1) -> 1
[0.00004911s] factorial(2) -> 2
[0.00008488s] factorial(3) -> 6
[0.00013208s] factorial(4) -> 24
[0.00019193s] factorial(5) -> 120
[0.00026107s] factorial(6) -> 720
6! = 720
```

工作原理

记得吗，如下代码：

```
@clock
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)
```

其实等价于：

```
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)

factorial = clock(factorial)
```

因此，在两个示例中，factorial 会作为 func 参数传给 clock（参见示例 7-15）。然后， clock 函数会返回 clocked 函数，Python 解释器在背后会把 clocked 赋值给 factorial。其实，导入 clockdeco_demo 模块后查看 factorial 的 __name__ 属性，会得到如下结果：

```
>>> import clockdeco_demo
>>> clockdeco_demo.factorial.__name__
'clocked'
>>>
```

所以，现在 factorial 保存的是 clocked 函数的引用。自此之后，每次调用 factorial(n)，执行的都是 clocked(n)。clocked 大致做了下面几件事。

(1) 记录初始时间 t0。

(2) 调用原来的 factorial 函数，保存结果。

(3) 计算经过的时间。

(4) 格式化收集的数据，然后打印出来。

(5) 返回第 2 步保存的结果。

这是装饰器的典型行为：把被装饰的函数替换成新函数，二者接受相同的参数，而且（通常）返回被装饰的函数本该返回的值，同时还会做些额外操作。

Gamma 等人写的《设计模式：可复用面向对象软件的基础》一书是这样概述“装饰器”模式的：“动态地给一个对象添加一些额外的职责。”函数装饰器符合这一说法。但是，在实现层面，Python 装饰器与《设计模式：可复用面向对象软件的基础》中所述的“装饰器”没有多少相似之处。“杂谈”会进一步探讨这个话题。

示例 7-15 中实现的 clock 装饰器有几个缺点：不支持关键字参数，而且遮盖了被装饰函数的 __name__ 和 __doc__ 属性。示例 7-17 使用

functools.wraps 装饰器把相关的属性从 func 复制到 clocked 中。此外，这个新版还能正确处理关键字参数。

示例 **7-17**　改进后的 clock 装饰器

```
# clockdeco2.py

import time
import functools

def clock(func):
    @functools.wraps(func)
    def clocked(*args, **kwargs):
        t0 = time.time()
        result = func(*args, **kwargs)
        elapsed = time.time() - t0
        name = func.__name__
        arg_lst = []
        if args:
            arg_lst.append(', '.join(repr(arg) for arg in args))
        if kwargs:
            pairs = ['%s=%r' % (k, w) for k, w in sorted(kwargs.items())]
            arg_lst.append(', '.join(pairs))
        arg_str = ', '.join(arg_lst)
        print('[%0.8fs] %s(%s) -> %r ' % (elapsed, name, arg_str, result))
        return result
    return clocked
```

functools.wraps 只是标准库中拿来即用的装饰器之一。下一节将介绍 functools 模块中最让人印象深刻的两个装饰器：lru_cache 和 singledispatch。

7.8　标准库中的装饰器

Python 内置了三个用于装饰方法的函数：`property`、`classmethod` 和 `staticmethod`。`property` 在 19.2 节讨论，另外两个在 9.4 节讨论。

另一个常见的装饰器是 `functools.wraps`，它的作用是协助构建行为良好的装饰器。我们在示例 7-17 中用过。标准库中最值得关注的两个装饰器是 `lru_cache` 和全新的 `singledispatch`（Python 3.4 新增）。这两个装饰器都在 `functools` 模块中定义。接下来分别讨论它们。

7.8.1　使用functools.lru_cache做备忘

`functools.lru_cache` 是非常实用的装饰器，它实现了备忘（memoization）功能。这是一项优化技术，它把耗时的函数的结果保存起来，避免传入相同的参数时重复计算。LRU 三个字母是“Least Recently Used”的缩写，表明缓存不会无限制增长，一段时间不用的缓存条目会被扔掉。

生成第 *n* 个斐波纳契数这种慢速递归函数适合使用 `lru_cache`，如示例 7-18 所示。

示例 **7-18**　生成第 *n* 个斐波纳契数，递归方式非常耗时

```
from clockdeco import clock

@clock
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-2) + fibonacci(n-1)

if __name__=='__main__':
    print(fibonacci(6))
```

运行 fibo_demo.py 得到的结果如下。除了最后一行，其余输出都是 `clock` 装饰器生成的。

```
$ python3 fibo_demo.py
[0.00000095s] fibonacci(0) -> 0
[0.00000095s] fibonacci(1) -> 1
[0.00007892s] fibonacci(2) -> 1
[0.00000095s] fibonacci(1) -> 1
[0.00000095s] fibonacci(0) -> 0
[0.00000095s] fibonacci(1) -> 1
[0.00003815s] fibonacci(2) -> 1
[0.00007391s] fibonacci(3) -> 2
[0.00018883s] fibonacci(4) -> 3
[0.00000000s] fibonacci(1) -> 1
[0.00000095s] fibonacci(0) -> 0
[0.00000119s] fibonacci(1) -> 1
[0.00004911s] fibonacci(2) -> 1
[0.00009704s] fibonacci(3) -> 2
[0.00000000s] fibonacci(0) -> 0
[0.00000000s] fibonacci(1) -> 1
[0.00002694s] fibonacci(2) -> 1
[0.00000095s] fibonacci(1) -> 1
[0.00000095s] fibonacci(0) -> 0
[0.00000095s] fibonacci(1) -> 1
[0.00005102s] fibonacci(2) -> 1
[0.00008917s] fibonacci(3) -> 2
[0.00015593s] fibonacci(4) -> 3
[0.00029993s] fibonacci(5) -> 5
[0.00052810s] fibonacci(6) -> 8
8
```

浪费时间的地方很明显：`fibonacci(1)` 调用了 8 次，`fibonacci(2)` 调用了 5 次……但是，如果增加两行代码，使用 `lru_cache`，性能会显著改善，如示例 7-19 所示。

示例 **7-19**　使用缓存实现，速度更快

```
import functools

from clockdeco import clock

@functools.lru_cache() # ❶
@clock  # ❷
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-2) + fibonacci(n-1)
```

```
if __name__=='__main__':
    print(fibonacci(6))
```

❶ 注意，必须像常规函数那样调用 lru_cache。这一行中有一对括号：@functools.lru_cache()。这么做的原因是，lru_cache 可以接受配置参数，稍后说明。

❷ 这里叠放了装饰器：@lru_cache() 应用到 @clock 返回的函数上。

这样一来，执行时间减半了，而且 n 的每个值只调用一次函数：

```
$ python3 fibo_demo_lru.py
[0.00000119s] fibonacci(0) -> 0
[0.00000119s] fibonacci(1) -> 1
[0.00010800s] fibonacci(2) -> 1
[0.00000787s] fibonacci(3) -> 2
[0.00016093s] fibonacci(4) -> 3
[0.00001216s] fibonacci(5) -> 5
[0.00025296s] fibonacci(6) -> 8
```

在计算 fibonacci(30) 的另一个测试中，示例 7-19 中的版本在 0.0005 秒内调用了 31 次 fibonacci 函数，而示例 7-18 中未缓存的版本调用 fibonacci 函数 2 692 537 次，在使用 Intel Core i7 处理器的笔记本电脑中耗时 17.7 秒。

除了优化递归算法之外，lru_cache 在从 Web 中获取信息的应用中也能发挥巨大作用。

特别要注意，lru_cache 可以使用两个可选的参数来配置。它的签名是：

```
functools.lru_cache(maxsize=128, typed=False)
```

maxsize 参数指定存储多少个调用的结果。缓存满了之后，旧的结果会被扔掉，腾出空间。为了得到最佳性能，maxsize 应该设为 2 的幂。typed 参数如果设为 True，把不同参数类型得到的结果分开保

存，即把通常认为相等的浮点数和整数参数（如 1 和 1.0）区分开。顺便说一下，因为 lru_cache 使用字典存储结果，而且键根据调用时传入的定位参数和关键字参数创建，所以被 lru_cache 装饰的函数，它的所有参数都必须是可散列的。

接下来讨论吸引人的 functools.singledispatch 装饰器。

7.8.2　单分派泛函数

假设我们在开发一个调试 Web 应用的工具，我们想生成 HTML，显示不同类型的 Python 对象。

我们可能会编写这样的函数：

```
import html

def htmlize(obj):
    content = html.escape(repr(obj))
    return '<pre>{}</pre>'.format(content)
```

这个函数适用于任何 Python 类型，但是现在我们想做个扩展，让它使用特别的方式显示某些类型。

- str：把内部的换行符替换为 '
\n'；不使用 <pre>，而是使用 <p>。
- int：以十进制和十六进制显示数字。
- list：输出一个 HTML 列表，根据各个元素的类型进行格式化。

我们想要的行为如示例 7-20 所示。

示例 **7-20**　生成 HTML 的 htmlize 函数，调整了几种对象的输出

```
>>> htmlize({1, 2, 3})  ❶
'<pre>{1, 2, 3}</pre>'
>>> htmlize(abs)
'<pre><built-in function abs></pre>'
```

```
>>> htmlize('Heimlich & Co.\n- a game')  ❷
'<p>Heimlich & Co.<br>\n- a game</p>'
>>> htmlize(42)  ❸
'<pre>42 (0x2a)</pre>'
>>> print(htmlize(['alpha', 66, {3, 2, 1}]))  ❹
<ul>
<li><p>alpha</p></li>
<li><pre>66 (0x42)</pre></li>
<li><pre>{1, 2, 3}</pre></li>
</ul>
```

❶ 默认情况下，在 `<pre></pre>` 中显示 HTML 转义后的对象字符串表示形式。

❷ 为 `str` 对象显示的也是 HTML 转义后的字符串表示形式，不过放在 `<p></p>` 中，而且使用 `<br>` 表示换行。

❸ `int` 显示为十进制和十六进制两种形式，放在 `<pre></pre>` 中。

❹ 各个列表项目根据各自的类型格式化，整个列表则渲染成 HTML 列表。

因为 Python 不支持重载方法或函数，所以我们不能使用不同的签名定义 `htmlize` 的变体，也无法使用不同的方式处理不同的数据类型。在 Python 中，一种常见的做法是把 `htmlize` 变成一个分派函数，使用一串 `if/elif/elif`，调用专门的函数，如 `htmlize_str`、`htmlize_int`，等等。这样不便于模块的用户扩展，还显得笨拙：时间一长，分派函数 `htmlize` 会变得很大，而且它与各个专门函数之间的耦合也很紧密。

Python 3.4 新增的 `functools.singledispatch` 装饰器可以把整体方案拆分成多个模块，甚至可以为你无法修改的类提供专门的函数。使用 `@singledispatch` 装饰的普通函数会变成泛函数（generic function）：根据第一个参数的类型，以不同方式执行相同操作的一组函数。[4] 具体做法参见示例 7-21。

[4]这才称得上是单分派。如果根据多个参数选择专门的函数，那就是多分派了。

functools.singledispatch 是 Python 3.4 增加的，PyPI 中的 singledispatch 包（https://pypi.python.org/pypi/singledispatch）可以向后兼容 Python 2.6 到 Python 3.3。

示例 **7-21** singledispatch 创建一个自定义的 htmlize.register 装饰器，把多个函数绑在一起组成一个泛函数

```
from functools import singledispatch
from collections import abc
import numbers
import html

@singledispatch  ❶
def htmlize(obj):
    content = html.escape(repr(obj))
    return '<pre>{}</pre>'.format(content)

@htmlize.register(str)  ❷
def _(text):            ❸
    content = html.escape(text).replace('\n', '<br>\n')
    return '<p>{0}</p>'.format(content)

@htmlize.register(numbers.Integral)  ❹
def _(n):
    return '<pre>{0} (0x{0:x})</pre>'.format(n)

@htmlize.register(tuple)  ❺
@htmlize.register(abc.MutableSequence)
def _(seq):
    inner = '</li>\n<li>'.join(htmlize(item) for item in seq)
    return '<ul>\n<li>' + inner + '</li>\n</ul>'
```

❶ @singledispatch 标记处理 object 类型的基函数。

❷ 各个专门函数使用 @«base_function».register(«type») 装饰。

❸ 专门函数的名称无关紧要；_ 是个不错的选择，简单明了。

❹ 为每个需要特殊处理的类型注册一个函数。numbers.Integral 是 int 的虚拟超类。

❺ 可以叠放多个 `register` 装饰器，让同一个函数支持不同类型。

只要可能，注册的专门函数应该处理抽象基类（如 `numbers.Integral` 和 `abc.MutableSequence`），不要处理具体实现（如 `int` 和 `list`）。这样，代码支持的兼容类型更广泛。例如，Python 扩展可以子类化 `numbers.Integral`，使用固定的位数实现 `int` 类型。

> 使用抽象基类检查类型，可以让代码支持这些抽象基类现有和未来的具体子类或虚拟子类。抽象基类的作用和虚拟子类的概念在第 11 章讨论。

`singledispatch` 机制的一个显著特征是，你可以在系统的任何地方和任何模块中注册专门函数。如果后来在新的模块中定义了新的类型，可以轻松地添加一个新的专门函数来处理那个类型。此外，你还可以为不是自己编写的或者不能修改的类添加自定义函数。

`singledispatch` 是经过深思熟虑之后才添加到标准库中的，它提供的特性很多，这里无法一一说明。这个机制最好的文档是“PEP 443 — Single-dispatch generic functions”（https://www.python.org/dev/peps/pep-0443/）。

> `@singledispatch` 不是为了把 Java 的那种方法重载带入 Python。在一个类中为同一个方法定义多个重载变体，比在一个函数中使用一长串 `if/elif/elif/elif` 块要更好。但是这两种方案都有缺陷，因为它们让代码单元（类或函数）承担的职责太多。`@singledispath` 的优点是支持模块化扩展：各个模块可以为它支持的各个类型注册一个专门函数。

装饰器是函数，因此可以组合起来使用（即，可以在已经被装饰的函数上应用装饰器，如示例 7-21 所示）。下一节说明其中的原理。

7.9　叠放装饰器

示例 7-19 演示了叠放装饰器的方式：`@lru_cache` 应用到 `@clock` 装饰 `fibonacci` 得到的结果上。在示例 7-21 中，模块中最后一个函数应用了两个 `@htmlize.register` 装饰器。

把 `@d1` 和 `@d2` 两个装饰器按顺序应用到 `f` 函数上，作用相当于 `f = d1(d2(f))`。

也就是说，下述代码：

```
@d1
@d2
def f():
    print('f')
```

等同于：

```
def f():
    print('f')

f = d1(d2(f))
```

除了叠放装饰器之外，本章还用到了几个接受参数的装饰器，例如 `@lru_cache()` 和示例 7-21 中 `@singledispatch` 生成的 `htmlize.register(«type»)`。下一节说明如何构建接受参数的装饰器。

7.10　参数化装饰器

解析源码中的装饰器时，Python 把被装饰的函数作为第一个参数传给装饰器函数。那怎么让装饰器接受其他参数呢？答案是：创建一个装饰器工厂函数，把参数传给它，返回一个装饰器，然后再把它应用到要装饰的函数上。不明白什么意思？当然。下面以我们见过的最简单的装饰器为例说明：示例 7-22 中的 `register`。

示例 **7-22**　示例 7-2 中 registration.py 模块的删减版，这里再次给出是为了便于讲解

```
registry = []

def register(func):
    print('running register(%s)' % func)
    registry.append(func)
    return func

@register
def f1():
    print('running f1()')

print('running main()')
print('registry ->', registry)
f1()
```

7.10.1　一个参数化的注册装饰器

为了便于启用或禁用 `register` 执行的函数注册功能，我们为它提供一个可选的 `active` 参数，设为 `False` 时，不注册被装饰的函数。实现方式参见示例 7-23。从概念上看，这个新的 `register` 函数不是装饰器，而是装饰器工厂函数。调用它会返回真正的装饰器，这才是应用到目标函数上的装饰器。

示例 **7-23**　为了接受参数，新的 `register` 装饰器必须作为函数调用

```
registry = set()  ❶
def register(active=True):  ❷
    def decorate(func):  ❸
        print('running register(active=%s)->decorate(%s)'
              % (active, func))
        if active:   ❹
            registry.add(func)
        else:
            registry.discard(func)  ❺

        return func  ❻
    return decorate  ❼

@register(active=False)  ❽
def f1():
    print('running f1()')

@register()  ❾
def f2():
    print('running f2()')

def f3():
    print('running f3()')
```

❶ `registry` 现在是一个 `set` 对象，这样添加和删除函数的速度更快。

❷ `register` 接受一个可选的关键字参数。

❸ `decorate` 这个内部函数是真正的装饰器；注意，它的参数是一个函数。

❹ 只有 `active` 参数的值（从闭包中获取）是 `True` 时才注册 `func`。

❺ 如果 `active` 不为真，而且 `func` 在 `registry` 中，那么把它删除。

❻ `decorate` 是装饰器，必须返回一个函数。

❼ `register` 是装饰器工厂函数，因此返回 `decorate`。

❽ `@register` 工厂函数必须作为函数调用，并且传入所需的参数。

❾ 即使不传入参数，`register` 也必须作为函数调用

（@register()），即要返回真正的装饰器 decorate。

这里的关键是，register() 要返回 decorate，然后把它应用到被装饰的函数上。

示例 7-23 中的代码在 registration_param.py 模块中。如果导入，得到的结果如下：

```
>>> import registration_param
running register(active=False)->decorate(<function f1 at 0x10063c1e0>)
running register(active=True)->decorate(<function f2 at 0x10063c268>)
>>> registration_param.registry
{<function f2 at 0x10063c268>}
```

注意，只有 f2 函数在 registry 中；f1 不在其中，因为传给 register 装饰器工厂函数的参数是 active=False，所以应用到 f1 上的 decorate 没有把它添加到 registry 中。

如果不使用 @ 句法，那就要像常规函数那样使用 register；若想把 f 添加到 registry 中，则装饰 f 函数的句法是 register()(f)；不想添加（或把它删除）的话，句法是 register(active=False)(f)。示例 7-24 演示了如何把函数添加到 registry 中，以及如何从中删除函数。

示例 **7-24**　使用示例 7-23 中的 registration_param 模块

```
>>> from registration_param import *
running register(active=False)->decorate(<function f1 at 0x10073c1e0>)
running register(active=True)->decorate(<function f2 at 0x10073c268>)
>>> registry  # ❶
{<function f2 at 0x10073c268>}
>>> register()(f3)  # ❷
running register(active=True)->decorate(<function f3 at 0x10073c158>)
<function f3 at 0x10073c158>
>>> registry  # ❸
{<function f3 at 0x10073c158>, <function f2 at 0x10073c268>}
>>> register(active=False)(f2)  # ❹
running register(active=False)->decorate(<function f2 at 0x10073c268>)
<function f2 at 0x10073c268>
>>> registry  # ❺
{<function f3 at 0x10073c158>}
```

❶ 导入这个模块时，f2 在 registry 中。

❷ register() 表达式返回 decorate，然后把它应用到 f3 上。

❸ 前一行把 f3 添加到 registry 中。

❹ 这次调用从 registry 中删除 f2。

❺ 确认 registry 中只有 f3。

参数化装饰器的原理相当复杂，我们刚刚讨论的那个比大多数都简单。参数化装饰器通常会把被装饰的函数替换掉，而且结构上需要多一层嵌套。接下来会探讨这种函数金字塔。

7.10.2　参数化clock装饰器

本节再次探讨 clock 装饰器，为它添加一个功能：让用户传入一个格式字符串，控制被装饰函数的输出。参见示例 7-25。

> 为了简单起见，示例 7-25 基于示例 7-15 中最初实现的 clock，而不是示例 7-17 中使用 @functools.wraps 改进后的版本，因为那一版增加了一层函数。

示例 **7-25**　clockdeco_param.py 模块：参数化 clock 装饰器

```
import time

DEFAULT_FMT = '[{elapsed:0.8f}s] {name}({args}) -> {result}'

def clock(fmt=DEFAULT_FMT):   ❶
    def decorate(func):       ❷
        def clocked(*_args):  ❸
            t0 = time.time()
            _result = func(*_args)  ❹
            elapsed = time.time() - t0
            name = func.__name__
            args = ', '.join(repr(arg) for arg in _args)  ❺
```

```
            result = repr(_result)  ❻
            print(fmt.format(**locals()))  ❼
            return _result  ❽
        return clocked  ❾
    return decorate  ❿

if __name__ == '__main__':

    @clock()
    def snooze(seconds):
        time.sleep(seconds)

    for i in range(3):
        snooze(.123)
```

❶ clock 是参数化装饰器工厂函数。

❷ decorate 是真正的装饰器。

❸ clocked 包装被装饰的函数。

❹ _result 是被装饰的函数返回的真正结果。

❺ _args 是 clocked 的参数，args 是用于显示的字符串。

❻ result 是 _result 的字符串表示形式，用于显示。

❼ 这里使用 **locals() 是为了在 fmt 中引用 clocked 的局部变量。

❽ clocked 会取代被装饰的函数，因此它应该返回被装饰的函数返回的值。

❾ decorate 返回 clocked。

❿ clock 返回 decorate。

⓫ 在这个模块中测试，不传入参数调用 clock()，因此应用的装饰器使用默认的格式 str。

在 shell 中运行示例 7-25，会得到下述结果：

```
$ python3 clockdeco_param.py
[0.12412500s] snooze(0.123) -> None
[0.12411904s] snooze(0.123) -> None
[0.12410498s] snooze(0.123) -> None
```

示例 7-26 和示例 7-27 是另外两个模块，它们使用了 `clockdeco_param` 模块中的新功能，随后是两个模块输出的结果。

示例 **7-26**　clockdeco_param_demo1.py

```
import time
from clockdeco_param import clock

@clock('{name}: {elapsed}s')
def snooze(seconds):
    time.sleep(seconds)

for i in range(3):
    snooze(.123)
```

示例 7-26 的输出：

```
$ python3 clockdeco_param_demo1.py
snooze: 0.12414693832397461s
snooze: 0.1241159439086914s
snooze: 0.12412118911743164s
```

示例 **7-27**　clockdeco_param_demo2.py

```
import time
from clockdeco_param import clock

@clock('{name}({args}) dt={elapsed:0.3f}s')
def snooze(seconds):
    time.sleep(seconds)

for i in range(3):
    snooze(.123)
```

示例 7-27 的输出：

```
$ python3 clockdeco_param_demo2.py
snooze(0.123) dt=0.124s
snooze(0.123) dt=0.124s
snooze(0.123) dt=0.124s
```

受本书篇幅限制，我们对装饰器的探讨到此结束。延伸阅读中的资料讨论了构建工业级装饰器的技术，尤其是 Graham Dumpleton 的博客和 `wrapt` 模块。

> Graham Dumpleton 和 Lennart Regebro（本书的技术审校之一）认为，装饰器最好通过实现 `__call__` 方法的类实现，不应该像本章的示例那样通过函数实现。我同意使用他们建议的方式实现非平凡的装饰器更好，但是使用函数解说这个语言特性的基本思想更易于理解。

7.11　本章小结

本章介绍了很多基础知识，虽然学习之路崎岖不平，我还是尽可能让路途平坦顺畅。毕竟，我们已经进入元编程领域了。

开始，我们先编写了一个没有内部函数的 `@register` 装饰器；最后，我们实现了有两层嵌套函数的参数化装饰器 `@clock()`。

尽管注册装饰器在多数情况下都很简单，但是在高级的 Python 框架中却有用武之地。我们使用注册方式对第 6 章的“策略”模式做了重构。

参数化装饰器基本上都涉及至少两层嵌套函数，如果想使用 `@functools.wraps` 生成装饰器，为高级技术提供更好的支持，嵌套层级可能还会更深，比如前面简要介绍过的叠放装饰器。

我们还讨论了标准库中 `functools` 模块提供的两个出色的函数装饰器：`@lru_cache()` 和 `@singledispatch`。

若想真正理解装饰器，需要区分导入时和运行时，还要知道变量作用域、闭包和新增的 `nonlocal` 声明。掌握闭包和 `nonlocal` 不仅对构建装饰器有帮助，还能协助你在构建 GUI 程序时面向事件编程，或者使用回调处理异步 I/O。

7.12 延伸阅读

《Python Cookbook（第 3 版）中文版》（David Beazley 和 Brian K. Jones 著）的第 9 章“元编程”有几个诀窍构建了基本的装饰器和特别复杂的装饰器。其中，“9.6 定义一个能接收可选参数的装饰器”一节中的装饰器可以作为常规的装饰器调用，也可以作为装饰器工厂函数调用，例如 `@clock` 或 `@clock()`。

Graham Dumpleton 写了一系列博客文章（https://github.com/GrahamDumpleton/wrapt/blob/develop/blog/README.md）深入剖析了如何实现行为良好的装饰器，第一篇是“How You Implemented Your Python Decorator is Wrong”（https://github.com/GrahamDumpleton/wrapt/blob/develop/blog/01-how-you-implemented-your-python-decorator-is-wrong.md）。他在这方面的深厚知识充分体现在在他编写的 `wrapt` 模块（http://wrapt.readthedocs.org/en/latest/）中。这个模块的作用是简化装饰器和动态函数包装器的实现，即使多层装饰也支持内省，而且行为正确，既可以应用到方法上，也可以作为描述符使用。（描述符在本书第 20 章讨论。）

Michele Simionato 开发了一个包，根据文档，它旨在“简化普通程序员使用装饰器的方式，并且通过各种复杂的示例推广装饰器”。这个包是 decorator（https://pypi.python.org/pypi/decorator），可通过 PyPI 安装。

Python Decorator Library 维基页面（https://wiki.python.org/moin/PythonDecoratorLibrary）在 Python 刚添加装饰器这个特性时就创建了，里面有很多示例。由于那个页面是几年前开始编写的，有些技术已经过时了，不过仍是很棒的灵感来源。

PEP 443（http://www.python.org/dev/peps/pep-0443/）对单分派泛函数的基本原理和细节做了说明。Guido van Rossum 很久以前（2005 年 3 月）写的一篇博客文章“Five-Minute Multimethods in Python”（http://www.artima.com/weblogs/viewpost.jsp?thread=101605）详细说明了如何使用装饰器实现泛函数（也叫多方法）。他给出的代码支持多分派（即根据多个定位参数进行分派）。Guido 写的多方法代码很

棒，但那只是教学示例。如果想使用现代的技术实现多分派泛函数，并支持在生产环境中使用，可以用 Martijn Faassen 开发的 Reg（http://reg.readthedocs.io/en/latest/）。Martijn 还是模型驱动型 REST 式 Web 框架 Morepath（http://morepath.readthedocs.org/en/latest/）的开发者。

Fredrik Lundh 写的一篇短文“Closures in Python”（http://effbot.org/zone/closure.htm）解说了闭包这个术语。

“PEP 3104—Access to Names in Outer Scopes”（http://www.python.org/dev/peps/pep-3104/）说明了引入 `nonlocal` 声明的原因：重新绑定既不在本地作用域中也不在全局作用域中的名称。这份 PEP 还概述了其他动态语言（Perl、Ruby、JavaScript，等等）解决这个问题的方式，以及 Python 中可用设计方案的优缺点。

“PEP 227—Statically Nested Scopes”（http://www.python.org/dev/peps/pep-0227/）更偏重于理论，说明了 Python 2.1 引入的词法作用域。词法作用域在这一版里是一种方案，到 Python 2.2 就变成了标准。此外，这份 PEP 还说明了 Python 中闭包的基本原理和实现方式的选择。

杂谈

任何把函数当作一等对象的语言，它的设计者都要面对一个问题：作为一等对象的函数在某个作用域中定义，但是可能会在其他作用域中调用。问题是，如何计算自由变量？首先出现的最简单的处理方式是使用“动态作用域”。也就是说，根据函数调用所在的环境计算自由变量。

如果 Python 使用动态作用域，不支持闭包，那么 `avg`（与示例 7-9 类似）可以写成这样：

```
>>> ### 这不是真实的Python控制台会话！ ###
>>> avg = make_averager()
>>> series = [] # ❶
>>> avg(10)
10.0
>>> avg(11) # ❷
10.5
```

```
>>> avg(12)
11.0
>>> series = [1] # ❸
>>> avg(5)
3.0
```

❶ 使用 avg 之前要自己定义 series = []，因此我们必须知道 averager（在 make_averager 内部）引用的是一个列表。

❷ 在背后使用 series 累计要计入平均值的值。

❸ 执行 series = [1] 后，之前的列表消失了。同时计算两个独立的移动平均值时可能会发生这种意外。

函数应该是黑盒，把实现隐藏起来，不让用户知道。但是对动态作用域来说，如果函数使用自由变量，程序员必须知道函数的内部细节，这样才能搭建正确运行所需的环境。

另一方面，动态作用域易于实现，这可能就是 John McCarthy 创建 Lisp（第一门把函数视作一等对象的语言）时采用这种方式的原因。Paul Graham 写的"The Roots of Lisp"一文（http://www.paulgraham.com/rootsoflisp.html）对 John McCarthy 关于 Lisp 语言那篇论文（"Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I"，http://www-formal.stanford.edu/jmc/recursive/recursive.html）做了通俗易懂的解说。McCarthy 那篇论文是和贝多芬第九交响曲一样伟大的杰作。Paul Graham 使用通俗易懂的语言翻译了那篇论文，把数学原理转换成了英语和可运行的代码。

Paul Graham 的注解还指出动态作用域难以实现。下面这段文字引自"The Roots of Lisp"一文：

> 就连第一个 Lisp 高阶函数示例都因为动态作用域而无法运行，这充分证明了动态作用域的危险性。McCarthy 在 1960 年可能没有全面认识到动态作用域的影响。动态作用域在各种 Lisp 实现中存在的时间特别长，直到 Sussman 和 Steele 在 1975 年开发出 Scheme 为止。词法作用域不会把 eval 的定义变得多么复杂，只是编译器可能更难编写。

如今，词法作用域已成常态：根据定义函数的环境计算自由变量。词法作用域让人更难实现支持一等函数的语言，因为需要支持闭包。不过，词法作用域让代码更易于阅读。Algol 之后出现的语言大都使用词法作用域。

多年来，Python 的 `lambda` 表达式不支持闭包，因此在博客圈的函数式编程极客群体中，这个特性的名声并不好。Python 2.2（2001 年 12 月发布）修正了这个问题，但是博客圈的固有印象不会轻易转变。自此之后，仅仅由于句法上的局限，`lambda` 一直处于尴尬的境地。

Python 装饰器和装饰器设计模式

Python 函数装饰器符合 Gamma 等人在《设计模式：可复用面向对象软件的基础》一书中对"装饰器"模式的一般描述："动态地给一个对象添加一些额外的职责。就扩展功能而言，装饰器模式比子类化更灵活。"

在实现层面，Python 装饰器与"装饰器"设计模式不同，但是有些相似之处。

在设计模式中，`Decorator` 和 `Component` 是抽象类。为了给具体组件添加行为，具体装饰器的实例要包装具体组件的实例。《设计模式：可复用面向对象软件的基础》一书是这样说的：

> 装饰器与它所装饰的组件接口一致，因此它对使用该组件的客户透明。它将客户请求转发给该组件，并且可能在转发前后执行一些额外的操作（例如绘制一个边框）。透明性使得你可以递归嵌套多个装饰器，从而可以添加任意多的功能。（第 115 页）

在 Python 中，装饰器函数相当于 `Decorator` 的具体子类，而装饰器返回的内部函数相当于装饰器实例。返回的函数包装了被装饰的函数，这相当于"装饰器"设计模式中的组件。返回的函数是透明的，因为它接受相同的参数，符合组件的接口。返回的函数把调用转发给组件，可以在转发前后执行额外的操作。因此，前面引用那段话的最后一句可以改成："透明性使得你可以递归嵌套多个装饰器，从而可以添加任意多的行为。"这就是叠放装饰器的理论基

础。

注意，我不是建议在 Python 程序中使用函数装饰器实现“装饰器”模式。在特定情况下确实可以这么做，但是一般来说，实现“装饰器”模式时最好使用类表示装饰器和要包装的组件。

第四部分　面向对象惯用法

第 8 章　对象引用、可变性和垃圾回收

> “你不开心，”白骑士用一种忧虑的声调说，“让我给你唱一首歌安慰你吧……这首歌的曲名叫作 :《黑线鳕的眼睛》。”
>
> “哦，那是一首歌的曲名，是吗？”爱丽丝问道，她试着使自己感到有兴趣。
>
> “不，你不明白，”白骑士说，看来有些心烦的样子，“那是人家这么叫的曲名。真正的曲名是《老而又老的老头儿》。”（改编自第 8 章“这是我自己的发明”）
>
> ——Lewis Carroll
> 《爱丽丝镜中奇遇记》

爱丽丝和白骑士为本章要讨论的内容定了基调。本章的主题是对象与对象名称之间的区别。名称不是对象，而是单独的东西。

本章先以一个比喻说明 Python 的变量：变量是标注，而不是盒子。如果你不知道引用式变量是什么，可以像这样对别人解释别名。

然后，本章讨论对象标识、值和别名等概念。随后，本章会揭露元组的一个神奇特性：元组是不可变的，但是其中的值可以改变，之后就引申到浅复制和深复制。接下来的话题是引用和函数参数：可变的参数默认值导致的问题，以及如何安全地处理函数的调用者传入的可变参数。

本章最后一节讨论垃圾回收、`del` 命令，以及如何使用弱引用“记住”对象，而无需对象本身存在。

本章的内容有点儿枯燥，但是这些话题却是解决 Python 程序中很多不易察觉的 bug 的关键。

首先，我们要抛弃变量是存储数据的盒子这一错误观念。

8.1　变量不是盒子

1997 年夏天，我在 MIT 学了一门 Java 课程。Lynn Andrea Stein 教授（一位获奖的计算机科学教育工作者，目前在欧林工程学院教书）指出，人们经常使用“变量是盒子”这样的比喻，但是这有碍于理解面向对象语言中的引用式变量。Python 变量类似于 Java 中的引用式变量，因此最好把它们理解为附加在对象上的标注。

在示例 8-1 所示的交互式控制台中，无法使用“变量是盒子”做解释。图 8-1 说明了在 Python 中为什么不能使用盒子比喻，而便利贴则指出了变量的正确工作方式。

示例 **8-1**　变量 `a` 和 `b` 引用同一个列表，而不是那个列表的副本

```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> b
[1, 2, 3, 4]
```

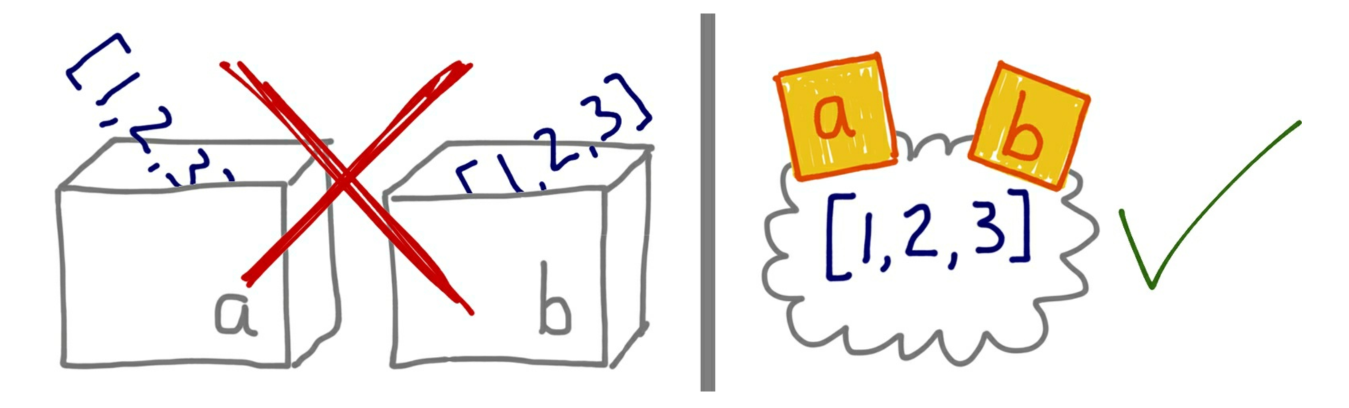


图 **8-1**：如果把变量想象为盒子，那么无法解释 **Python** 中的赋值；应该把变量视作便利贴，这样示例 **8-1** 中的行为就好解释了

Stein 教授还反复讲解了赋值方式。例如讲到 seesaw 对象时，她会说“把变量 *s* 分配给 seesaw”，绝不会说“把 seesaw 分配给变量 *s*”。对引用式变量来说，说把变量分配给对象更合理，反过来说就有问题。毕竟，对象在赋值之前就创建了。示例 8-2 证明赋值语句的右边先执行。

示例 **8-2**　创建对象之后才会把变量分配给对象

```
>>> class Gizmo:
...    def __init__(self):
...        print('Gizmo id: %d' % id(self))
...
>>> x = Gizmo()
Gizmo id: 4301489152  ❶
>>> y = Gizmo() * 10  ❷
Gizmo id: 4301489432  ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for *: 'Gizmo' and 'int'
>>>
>>> dir()  ❹
['Gizmo', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'x']
```

❶ 输出的 `Gizmo id: ...` 是创建 `Gizmo` 实例的副作用。

❷ 在乘法运算中使用 `Gizmo` 实例会抛出异常。

❸ 这里表明，在尝试求积之前其实会创建一个新的 `Gizmo` 实例。

❹ 但是，肯定不会创建变量 `y`，因为在对赋值语句的右边进行求值时抛出了异常。

为了理解 Python 中的赋值语句，应该始终先读右边。对象在右边创建或获取，在此之后左边的变量才会绑定到对象上，这就像为对象贴上标注。忘掉盒子吧！

因为变量只不过是标注，所以无法阻止为对象贴上多个标注。贴的多个标注，就是别名。参见下一节。

8.2　标识、相等性和别名

Lewis Carroll 是 Charles Lutwidge Dodgson 教授的笔名。Carroll 先生指的就是 Dodgson 教授，二者是同一个人。示例 8-3 用 Python 表达了这个概念。

示例 **8-3**　`charles` 和 `lewis` 指代同一个对象

```
>>> charles = {'name': 'Charles L. Dodgson', 'born': 1832}
>>> lewis = charles  ❶
>>> lewis is charles
True
>>> id(charles), id(lewis)  ❷
(4300473992, 4300473992)
>>> lewis['balance'] = 950  ❸
>>> charles
{'name': 'Charles L. Dodgson', 'balance': 950, 'born': 1832}
```

❶ `lewis` 是 `charles` 的别名。

❷ `is` 运算符和 `id` 函数确认了这一点。

❸ 向 `lewis` 中添加一个元素相当于向 `charles` 中添加一个元素。

然而，假如有冒充者（姑且叫他 Alexander Pedachenko 博士）生于 1832 年，声称他是 Charles L. Dodgson。这个冒充者的证件可能一样，但是 Pedachenko 博士不是 Dodgson 教授。这种情况如图 8-2 所示。

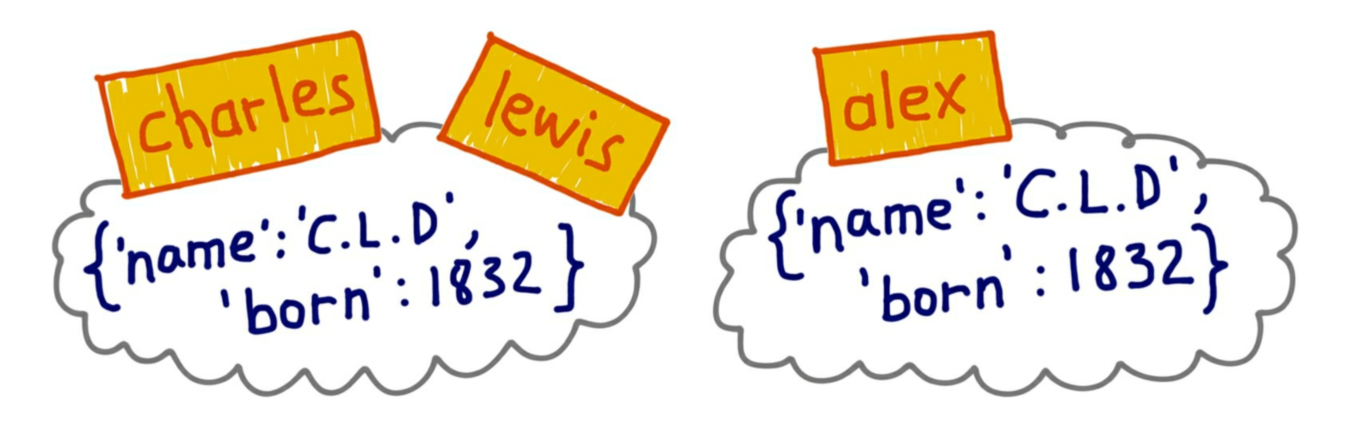


图 **8-2**：**`charles`** 和 **`lewis`** 绑定同一个对象，**`alex`** 绑定另一个具有相同内容的对象

示例 8-4 实现并测试了图 8-2 中那个 `alex` 对象。

示例 **8-4**　`alex` 与 `charles` 比较的结果是相等，但 `alex` 不是 `charles`

```
>>> alex = {'name': 'Charles L. Dodgson', 'born': 1832, 'balance': 950}  ❶
>>> alex == charles  ❷
True
>>> alex is not charles  ❸
True
```

❶ `alex` 指代的对象与赋值给 `charles` 的对象内容一样。

❷ 比较两个对象，结果相等，这是因为 `dict` 类的 `__eq__` 方法就是这样实现的。

❸ 但它们是不同的对象。这是 Python 说明标识不同的方式：`a is not b`。

示例 8-3 体现了别名。在那段代码中，`lewis` 和 `charles` 是别名，即两个变量绑定同一个对象。而 `alex` 不是 `charles` 的别名，因为二者绑定的是不同的对象。`alex` 和 `charles` 绑定的对象具有相同的值（`==` 比较的就是值），但是它们的标识不同。

Python 语言参考手册中的“3.1 Objects, values and types”一节（https://docs.python.org/3/reference/datamodel.html#objects-values-and-types）说道：

> 每个变量都有标识、类型和值。对象一旦创建，它的标识绝不会变；你可以把标识理解为对象在内存中的地址。is 运算符比较两个对象的标识；id() 函数返回对象标识的整数表示。

对象 ID 的真正意义在不同的实现中有所不同。在 CPython 中，`id()` 返回对象的内存地址，但是在其他 Python 解释器中可能是别的值。关键是，ID 一定是唯一的数值标注，而且在对象的生命周期中绝不会变。

其实，编程中很少使用 `id()` 函数。标识最常使用 `is` 运算符检查，而不是直接比较 ID。接下来讨论 `is` 和 `==` 的异同。

8.2.1　在==和is之间选择

`==` 运算符比较两个对象的值（对象中保存的数据），而 `is` 比较对象的标识。

通常，我们关注的是值，而不是标识，因此 Python 代码中 `==` 出现的频率比 `is` 高。

然而，在变量和单例值之间比较时，应该使用 `is`。目前，最常使用 `is` 检查变量绑定的值是不是 `None`。下面是推荐的写法：

```
x is None
```

否定的正确写法是：

```
x is not None
```

`is` 运算符比 `==` 速度快，因为它不能重载，所以 Python 不用寻找并调用特殊方法，而是直接比较两个整数 ID。而 `a == b` 是语法糖，等同于 `a.__eq__(b)`。继承自 `object` 的 `__eq__` 方法比较两个对象的 ID，结果与 `is` 一样。但是多数内置类型使用更有意义的方式覆盖了 `__eq__` 方法，会考虑对象属性的值。相等性测试可能涉及大量处理工作，例如，比较大型集合或嵌套层级深的结构时。

在结束对标识和相等性的讨论之前，我们来看看著名的不可变类型 `tuple`（元组），它没有你想象的那么一成不变。

8.2.2　元组的相对不可变性

元组与多数 Python 集合（列表、字典、集，等等）一样，保存的是对象的引用。[1] 如果引用的元素是可变的，即便元组本身不可变，元素依然可变。也就是说，元组的不可变性其实是指 `tuple` 数据结构的物理内

容（即保存的引用）不可变，与引用的对象无关。

> [1]而 str、bytes 和 array.array 等单一类型序列是扁平的，它们保存的不是引用，而是在连续的内存中保存数据本身（字符、字节和数字）。

示例 8-5 表明，元组的值会随着引用的可变对象的变化而变。元组中不可变的是元素的标识。

示例 **8-5**　一开始，`t1` 和 `t2` 相等，但是修改 `t1` 中的一个可变元素后，二者不相等了

```
>>> t1 = (1, 2, [30, 40])  ❶
>>> t2 = (1, 2, [30, 40])  ❷
>>> t1 == t2  ❸
True
>>> id(t1[-1])  ❹
4302515784
>>> t1[-1].append(99)  ❺
>>> t1
(1, 2, [30, 40, 99])
>>> id(t1[-1])  ❻
4302515784
>>> t1 == t2  ❼
False
```

❶ `t1` 不可变，但是 `t1[-1]` 可变。

❷ 构建元组 `t2`，它的元素与 `t1` 一样。

❸ 虽然 `t1` 和 `t2` 是不同的对象，但是二者相等——与预期相符。

❹ 查看 `t1[-1]` 列表的标识。

❺ 就地修改 `t1[-1]` 列表。

❻ `t1[-1]` 的标识没变，只是值变了。

❼ 现在，`t1` 和 `t2` 不相等。

元组的相对不可变性解释了 2.6.1 节的谜题。这也是有些元组不可散列（参见 3.1 节中的“什么是可散列的数据类型”附注栏）的原因。

复制对象时，相等性和标识之间的区别有更深入的影响。副本与源对象相等，但是 ID 不同。可是，如果对象中包含其他对象，那么应该复制内部对象吗？可以共享内部对象吗？这些问题没有唯一的答案。参见下述讨论。

8.3　默认做浅复制

复制列表（或多数内置的可变集合）最简单的方式是使用内置的类型构造方法。例如：

```
>>> l1 = [3, [55, 44], (7, 8, 9)]
>>> l2 = list(l1)  ❶
>>> l2
[3, [55, 44], (7, 8, 9)]
>>> l2 == l1  ❷
True
>>> l2 is l1  ❸
False
```

❶ `list(l1)` 创建 `l1` 的副本。

❷ 副本与源列表相等。

❸ 但是二者指代不同的对象。对列表和其他可变序列来说，还能使用简洁的 `l2 = l1[:]` 语句创建副本。

然而，构造方法或 `[:]` 做的是浅复制（即复制了最外层容器，副本中的元素是源容器中元素的引用）。如果所有元素都是不可变的，那么这样没有问题，还能节省内存。但是，如果有可变的元素，可能就会导致意想不到的问题。

在示例 8-6 中，我们为一个包含另一个列表和一个元组的列表做了浅复制，然后做了些修改，看看对引用的对象有什么影响。

> 如果你手头有联网的电脑，我强烈建议你在 Python Tutor 网站（http://www.pythontutor.com）中查看示例 8-6 的交互式动画。写作本书时，无法直接链接 pythontutor.com 中准备好的示例，不过这个工具很出色，因此值得花点时间复制粘贴代码。

示例 **8-6**　为一个包含另一个列表的列表做浅复制；把这段代码复制粘贴到 Python Tutor 网站中，看看动画效果

```
l1 = [3, [66, 55, 44], (7, 8, 9)]
l2 = list(l1)      # ❶
l1.append(100)     # ❷
l1[1].remove(55)   # ❸
print('l1:', l1)
print('l2:', l2)
l2[1] += [33, 22]  # ❹
l2[2] += (10, 11)  # ❺
print('l1:', l1)
print('l2:', l2)
```

❶ `l2` 是 `l1` 的浅复制副本。此时的状态如图 8-3 所示。

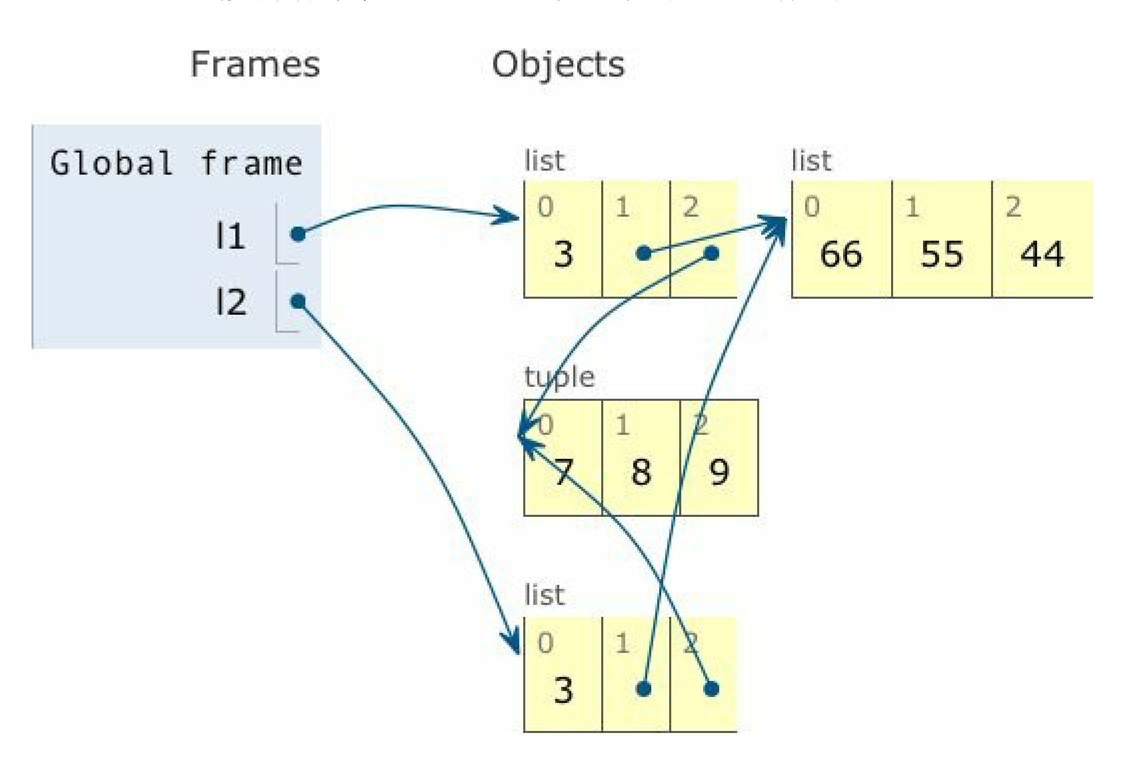


图 **8-3**：示例 **8-6** 执行 **`l2 = list(l1)`** 赋值后的程序状态。**`l1`** 和 **`l2`** 指代不同的列表，但是二者引用同一个列表 **`[66, 55, 44]`** 和元组 **`(7, 8, 9)`**（图表由 **Python Tutor** 网站生成）

❷ 把 `100` 追加到 `l1` 中，对 `l2` 没有影响。

❸ 把内部列表 `l1[1]` 中的 `55` 删除。这对 `l2` 有影响，因为 `l2[1]` 绑定

的列表与 `l1[1]` 是同一个。

❹ 对可变的对象来说，如 `l2[1]` 引用的列表，`+=` 运算符就地修改列表。这次修改在 `l1[1]` 中也有体现，因为它是 `l2[1]` 的别名。

❺ 对元组来说，`+=` 运算符创建一个新元组，然后重新绑定给变量 `l2[2]`。这等同于 `l2[2] = l2[2] + (10, 11)`。现在，`l1` 和 `l2` 中最后位置上的元组不是同一个对象。如图 8-4 所示。

示例 8-6 的输出在示例 8-7 中，对象的最终状态如图 8-4 所示。

示例 **8-7**　示例 8-6 的输出

```
l1: [3, [66, 44], (7, 8, 9), 100]
l2: [3, [66, 44], (7, 8, 9)]
l1: [3, [66, 44, 33, 22], (7, 8, 9), 100]
l2: [3, [66, 44, 33, 22], (7, 8, 9, 10, 11)]
```

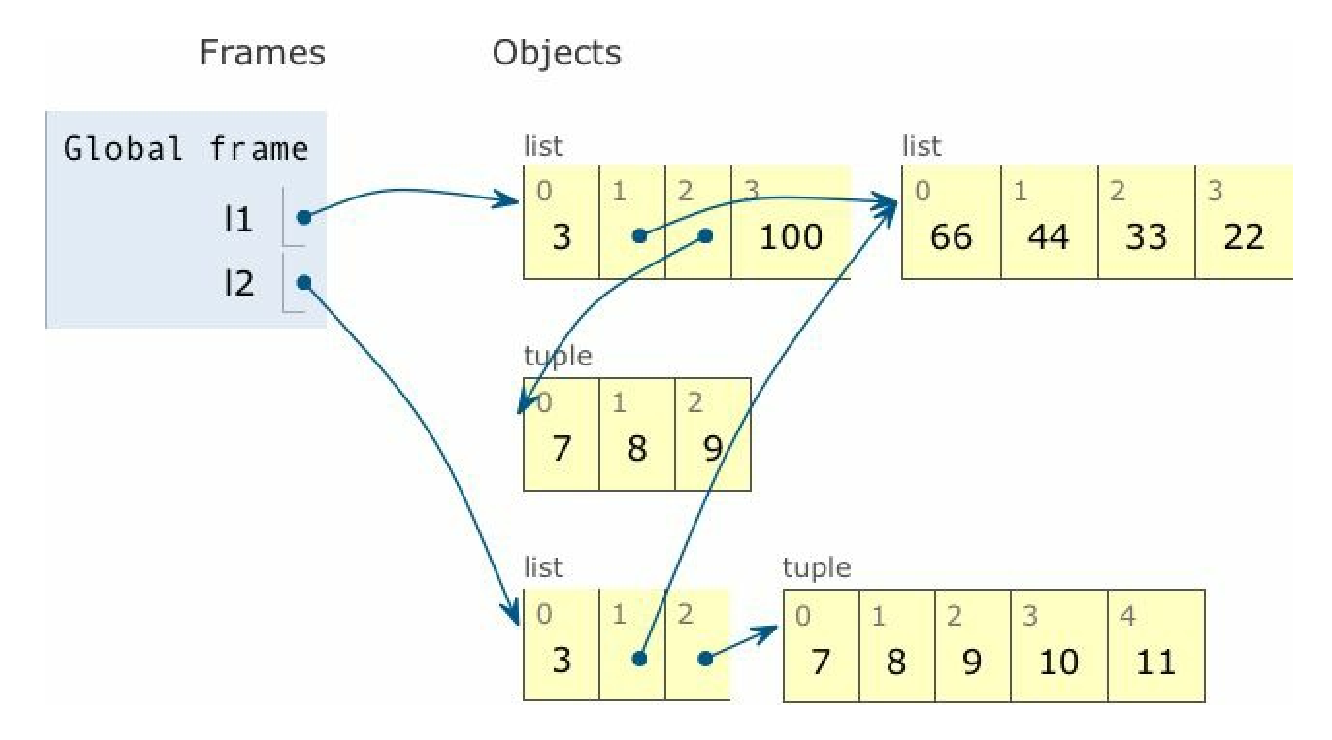


图 **8-4**：**`l1`** 和 **`l2`** 的最终状态：二者依然引用同一个列表对象，现在列表的值是 **`[66, 44, 33, 22]`**，不过 **`l2[2] += (10, 11)`** 创建一个新元组，内容是 **`(7, 8, 9, 10, 11)`**，它与 **`l1[2]`** 引用的元组 **`(7, 8, 9)`** 无关（图表由 **Python Tutor** 网站生成）

现在你应该明白了，浅复制容易操作，但是得到的结果可能并不是你想要的。接下来说明如何做深复制。

为任意对象做深复制和浅复制

浅复制没什么问题，但有时我们需要的是深复制（即副本不共享内部对象的引用）。copy 模块提供的 deepcopy 和 copy 函数能为任意对象做深复制和浅复制。

为了演示 copy() 和 deepcopy() 的用法，示例 8-8 定义了一个简单的类，Bus。这个类表示运载乘客的校车，在途中乘客会上车或下车。

示例 **8-8**　校车乘客在途中上车和下车

```
class Bus:

    def __init__(self, passengers=None):
        if passengers is None:
            self.passengers = []
        else:
            self.passengers = list(passengers)

    def pick(self, name):
        self.passengers.append(name)

    def drop(self, name):
        self.passengers.remove(name)
```

接下来，在示例 8-9 中的交互式控制台中，我们将创建一个 Bus 实例（bus1）和两个副本，一个是浅复制副本（bus2），另一个是深复制副本（bus3），看看在 bus1 有学生下车后会发生什么。

示例 **8-9**　使用 copy 和 deepcopy 产生的影响

```
>>> import copy
>>> bus1 = Bus(['Alice', 'Bill', 'Claire', 'David'])
>>> bus2 = copy.copy(bus1)
>>> bus3 = copy.deepcopy(bus1)
>>> id(bus1), id(bus2), id(bus3)
(4301498296, 4301499416, 4301499752)  ❶
```

```
>>> bus1.drop('Bill')
>>> bus2.passengers
['Alice', 'Claire', 'David']          ❷
>>> id(bus1.passengers), id(bus2.passengers), id(bus3.passengers)
(4302658568, 4302658568, 4302657800)  ❸
>>> bus3.passengers
['Alice', 'Bill', 'Claire', 'David']  ❹
```

❶ 使用 copy 和 deepcopy，创建 3 个不同的 Bus 实例。

❷ bus1 中的 'Bill' 下车后，bus2 中也没有他了。

❸ 审查 passengers 属性后发现，bus1 和 bus2 共享同一个列表对象，因为 bus2 是 bus1 的浅复制副本。

❹ bus3 是 bus1 的深复制副本，因此它的 passengers 属性指代另一个列表。

注意，一般来说，深复制不是件简单的事。如果对象有循环引用，那么这个朴素的算法会进入无限循环。deepcopy 函数会记住已经复制的对象，因此能优雅地处理循环引用，如示例 8-10 所示。

示例 **8-10**　循环引用：b 引用 a，然后追加到 a 中；deepcopy 会想办法复制 a

```
>>> a = [10, 20]
>>> b = [a, 30]
>>> a.append(b)
>>> a
[10, 20, [[...], 30]]
>>> from copy import deepcopy
>>> c = deepcopy(a)
>>> c
[10, 20, [[...], 30]]
```

此外，深复制有时可能太深了。例如，对象可能会引用不该复制的外部资源或单例值。我们可以实现特殊方法 __copy__() 和 __deepcopy__()，控制 copy 和 deepcopy 的行为，详情参见 copy 模块的文档（http://docs.python.org/3/library/copy.html）。

通过别名共享对象还能解释 Python 中传递参数的方式，以及使用可变类

型作为参数默认值引起的问题。接下来讨论这些问题。

8.4　函数的参数作为引用时

Python 唯一支持的参数传递模式是共享传参（call by sharing）。多数面向对象语言都采用这一模式，包括 Ruby、Smalltalk 和 Java（Java 的引用类型是这样，基本类型按值传参）。

共享传参指函数的各个形式参数获得实参中各个引用的副本。也就是说，函数内部的形参是实参的别名。

这种方案的结果是，函数可能会修改作为参数传入的可变对象，但是无法修改那些对象的标识（即不能把一个对象替换成另一个对象）。示例 8-11 中有个简单的函数，它在参数上调用 **+=** 运算符。分别把数字、列表和元组传给那个函数，实际传入的实参会以不同的方式受到影响。

示例 **8-11**　函数可能会修改接收到的任何可变对象

```
>>> def f(a, b):
...     a += b
...     return a
...
>>> x = 1
>>> y = 2
>>> f(x, y)
3
>>> x, y  ❶
(1, 2)
>>> a = [1, 2]
>>> b = [3, 4]
>>> f(a, b)
[1, 2, 3, 4]
>>> a, b  ❷
([1, 2, 3, 4], [3, 4])
>>> t = (10, 20)
>>> u = (30, 40)
>>> f(t, u)
(10, 20, 30, 40)
>>> t, u ❸
((10, 20), (30, 40))
```

❶ 数字 x 没变。

❷ 列表 a 变了。

❸ 元组 t 没变。

与函数参数相关的另一个问题是使用可变值作为默认值，下一节会讨论。

8.4.1　不要使用可变类型作为参数的默认值

可选参数可以有默认值，这是 Python 函数定义的一个很棒的特性，这样我们的 API 在进化的同时能保证向后兼容。然而，我们应该避免使用可变的对象作为参数的默认值。

下面在示例 8-12 中说明这个问题。我们以示例 8-8 中的 Bus 类为基础定义一个新类， HauntedBus，然后修改 __init__ 方法。这一次，passengers 的默认值不是 None，而是 []，这样就不用像之前那样使用 if 判断了。这个“聪明的举动”会让我们陷入麻烦。

示例 **8-12**　一个简单的类，说明可变默认值的危险

```
class HauntedBus:
    """备受幽灵乘客折磨的校车"""

def __init__(self, passengers=[]):  ❶
    self.passengers = passengers  ❷

def pick(self, name):
    self.passengers.append(name)  ❸

def drop(self, name):
    self.passengers.remove(name)
```

❶ 如果没传入 passengers 参数，使用默认绑定的列表对象，一开始是空列表。

❷ 这个赋值语句把 self.passengers 变成 passengers 的别名，而没有传入 passengers 参数时，后者又是默认列表的别名。

❸ 在 self.passengers 上调用 .remove() 和 .append() 方法时，修改的其实是默认列表，它是函数对象的一个属性。

HauntedBus 的诡异行为如示例 8-13 所示。

示例 **8-13**　备受幽灵乘客折磨的校车

```
>>> bus1 = HauntedBus(['Alice', 'Bill'])
>>> bus1.passengers
['Alice', 'Bill']
>>> bus1.pick('Charlie')
>>> bus1.drop('Alice')
>>> bus1.passengers  ❶
['Bill', 'Charlie']
>>> bus2 = HauntedBus()  ❷
>>> bus2.pick('Carrie')
>>> bus2.passengers
['Carrie']
>>> bus3 = HauntedBus()  ❸
>>> bus3.passengers  ❹
['Carrie']
>>> bus3.pick('Dave')
>>> bus2.passengers  ❺
['Carrie', 'Dave']
>>> bus2.passengers is bus3.passengers  ❻
True
>>> bus1.passengers  ❼
['Bill', 'Charlie']
```

❶ 目前没什么问题，bus1 没有出现异常。

❷ 一开始，bus2 是空的，因此把默认的空列表赋值给 self.passengers。

❸ bus3 一开始也是空的，因此还是赋值默认的列表。

❹ 但是默认列表不为空！

❺ 登上 bus3 的 Dave 出现在 bus2 中。

❻ 问题是，bus2.passengers 和 bus3.passengers 指代同一个列表。

❼ 但 bus1.passengers 是不同的列表。

问题在于，没有指定初始乘客的 HauntedBus 实例会共享同一个乘客列表。

这种问题很难发现。如示例 8-13 所示，实例化 HauntedBus 时，如果传入乘客，会按预期运作。但是不为 HauntedBus 指定乘客的话，奇怪的事就发生了，这是因为 self.passengers 变成了 passengers 参数默认值的别名。出现这个问题的根源是，默认值在定义函数时计算（通常在加载模块时），因此默认值变成了函数对象的属性。因此，如果默认值是可变对象，而且修改了它的值，那么后续的函数调用都会受到影响。

运行示例 8-13 中的代码之后，可以审查 HauntedBus.__init__ 对象，看看它的 __defaults__ 属性中的那些幽灵学生：

```
>>> dir(HauntedBus.__init__)  # doctest: +ELLIPSIS
['__annotations__', '__call__', ..., '__defaults__', ...]
>>> HauntedBus.__init__.__defaults__
(['Carrie', 'Dave'],)
```

最后，我们可以验证 bus2.passengers 是一个别名，它绑定到 HauntedBus.__init__.__defaults__ 属性的第一个元素上：

```
>>> HauntedBus.__init__.__defaults__[0] is bus2.passengers
True
```

可变默认值导致的这个问题说明了为什么通常使用 None 作为接收可变值的参数的默认值。在示例 8-8 中，__init__ 方法检查 passengers 参数的值是不是 None，如果是就把一个新的空列表赋值给 self.passengers。下一节会说明，如果 passengers 不是 None，正确的实现会把 passengers 的副本赋值给 self.passengers。下面详解。

8.4.2　防御可变参数

如果定义的函数接收可变参数，应该谨慎考虑调用方是否期望修改传入

的参数。

例如，如果函数接收一个字典，而且在处理的过程中要修改它，那么这个副作用要不要体现到函数外部？具体情况具体分析。这其实需要函数的编写者和调用方达成共识。

在本章最后一个校车示例中，`TwilightBus` 实例与客户共享乘客列表，这会产生意料之外的结果。在分析实现之前，我们先从客户的角度看看 `TwilightBus` 类是如何工作的。

示例 **8-14**　从 `TwilightBus` 下车后，乘客消失了

```
>>> basketball_team = ['Sue', 'Tina', 'Maya', 'Diana', 'Pat']  ❶
>>> bus = TwilightBus(basketball_team)  ❷
>>> bus.drop('Tina')  ❸
>>> bus.drop('Pat')
>>> basketball_team  ❹
['Sue', 'Maya', 'Diana']
```

❶ `basketball_team` 中有 5 个学生的名字。

❷ 使用这队学生实例化 `TwilightBus`。

❸ 一个学生从 `bus` 下车了，接着又有一个学生下车了。

❹ 下车的学生从篮球队中消失了！

`TwilightBus` 违反了设计接口的最佳实践，即“最少惊讶原则”。学生从校车中下车后，她的名字就从篮球队的名单中消失了，这确实让人惊讶。

示例 8-15 是 `TwilightBus` 的实现，随后解释了出现这个问题的原因。

示例 **8-15**　一个简单的类，说明接受可变参数的风险

```
class TwilightBus:
    """让乘客销声匿迹的校车"""

    def __init__(self, passengers=None):
        if passengers is None:
```

```
            self.passengers = []  ❶
        else:
            self.passengers = passengers  ❷

    def pick(self, name):
        self.passengers.append(name)

    def drop(self, name):
        self.passengers.remove(name)  ❸
```

❶ 这里谨慎处理，当 `passengers` 为 `None` 时，创建一个新的空列表。

❷ 然而，这个赋值语句把 `self.passengers` 变成 `passengers` 的别名，而后者是传给 `__init__` 方法的实参（即示例 8-14 中的 `basketball_team`）的别名。

❸ 在 `self.passengers` 上调用 `.remove()` 和 `.append()` 方法其实会修改传给构造方法的那个列表。

这里的问题是，校车为传给构造方法的列表创建了别名。正确的做法是，校车自己维护乘客列表。修正的方法很简单：在 `__init__` 中，传入 `passengers` 参数时，应该把参数值的副本赋值给 `self.passengers`，像示例 8-8 中那样做（8.3 节）。

```
def __init__(self, passengers=None):
    if passengers is None:
        self.passengers = []
    else:
        self.passengers = list(passengers)  ❶
```

❶ 创建 `passengers` 列表的副本；如果不是列表，就把它转换成列表。

在内部像这样处理乘客列表，就不会影响初始化校车时传入的参数了。此外，这种处理方式还更灵活：现在，传给 `passengers` 参数的值可以是元组或任何其他可迭代对象，例如 `set` 对象，甚至数据库查询结果，因为 `list` 构造方法接受任何可迭代对象。自己创建并管理列表可以确保支持所需的 `.remove()` 和 `.append()` 操作，这样 `.pick()` 和

.drop() 方法才能正常运作。

> 除非这个方法确实想修改通过参数传入的对象，否则在类中直接把参数赋值给实例变量之前一定要三思，因为这样会为参数对象创建别名。如果不确定，那就创建副本。这样客户会少些麻烦。

8.5　del和垃圾回收

> 对象绝不会自行销毁；然而，无法得到对象时，可能会被当作垃圾回收。
>
> —— Python 语言参考手册中“Data Model”一章

del 语句删除名称，而不是对象。del 命令可能会导致对象被当作垃圾回收，但是仅当删除的变量保存的是对象的最后一个引用，或者无法得到对象时。[2] 重新绑定也可能会导致对象的引用数量归零，导致对象被销毁。

> [2]如果两个对象相互引用，像示例 8-10 那样，当它们的引用只存在二者之间时，垃圾回收程序会判定它们都无法获取，进而把它们都销毁。

> 有个 `__del__` 特殊方法，但是它不会销毁实例，不应该在代码中调用。即将销毁实例时，Python 解释器会调用 `__del__` 方法，给实例最后的机会，释放外部资源。自己编写的代码很少需要实现 `__del__` 代码，有些 Python 新手会花时间实现，但却吃力不讨好，因为 `__del__` 很难用对。详情参见 Python 语言参考手册中“Data Model”一章中 `__del__` 特殊方法的文档（https://docs.python.org/3/reference/datamodel.html#object.__del__）。

在 CPython 中，垃圾回收使用的主要算法是引用计数。实际上，每个对象都会统计有多少引用指向自己。当引用计数归零时，对象立即就被销毁：CPython 会在对象上调用 `__del__` 方法（如果定义了），然后释放分配给对象的内存。CPython 2.0 增加了分代垃圾回收算法，用于检测引用循环中涉及的对象组——如果一组对象之间全是相互引用，即使再出色的引用方式也会导致组中的对象不可获取。Python 的其他实现有更复杂的垃圾回收程序，而且不依赖引用计数，这意味着，对象的引用数量为零时可能不会立即调用 `__del__` 方法。A. Jesse Jiryu Davis 写的“PyPy, Garbage Collection, and a Deadlock”一文（https://emptysqua.re/blog/pypy-garbage-collection-and-a-deadlock/）对 `__del__` 方法的恰当用法和不当用法做了讨论。

为了演示对象生命结束时的情形，示例 8-16 使用 `weakref.finalize` 注册一个回调函数，在销毁对象时调用。

示例 **8-16** 没有指向对象的引用时，监视对象生命结束时的情形

```
>>> import weakref
>>> s1 = {1, 2, 3}
>>> s2 = s1       ❶
>>> def bye():    ❷
...     print('Gone    with the wind...')
...
>>> ender = weakref.finalize(s1, bye)  ❸
>>> ender.alive  ❹
True
>>> del s1
>>> ender.alive  ❺
True
>>> s2 = 'spam'  ❻
Gone with the wind...
>>> ender.alive
False
```

❶ `s1` 和 `s2` 是别名，指向同一个集合，`{1, 2, 3}`。

❷ 这个函数一定不能是要销毁的对象的绑定方法，否则会有一个指向对象的引用。

❸ 在 `s1` 引用的对象上注册 `bye` 回调。

❹ 调用 `finalize` 对象之前，`.alive` 属性的值为 `True`。

❺ 如前所述，`del` 不删除对象，而是删除对象的引用。

❻ 重新绑定最后一个引用 `s2`，让 `{1, 2, 3}` 无法获取。对象被销毁了，调用了 `bye` 回调，`ender.alive` 的值变成了 `False`。

示例 8-16 的目的是明确指出 `del` 不会删除对象，但是执行 `del` 操作后可能会导致对象不可获取，从而被删除。

你可能觉得奇怪，为什么示例 8-16 中的 `{1, 2, 3}` 对象被销毁了？毕竟，我们把 `s1` 引用传给 `finalize` 函数了，而为了监控对象和调用回

调，必须要有引用。这是因为，`finalize` 持有 `{1, 2, 3}` 的弱引用，参见下一节。

8.6　弱引用

正是因为有引用，对象才会在内存中存在。当对象的引用数量归零后，垃圾回收程序会把对象销毁。但是，有时需要引用对象，而不让对象存在的时间超过所需时间。这经常用在缓存中。

弱引用不会增加对象的引用数量。引用的目标对象称为所指对象（referent）。因此我们说，弱引用不会妨碍所指对象被当作垃圾回收。

弱引用在缓存应用中很有用，因为我们不想仅因为被缓存引用着而始终保存缓存对象。

示例 8-17 展示了如何使用 `weakref.ref` 实例获取所指对象。如果对象存在，调用弱引用可以获取对象；否则返回 `None`。

> 示例 8-17 是一个控制台会话，Python 控制台会自动把 `_` 变量绑定到结果不为 `None` 的表达式结果上。这对我想演示的行为有影响，不过却凸显了一个实际问题：微观管理内存时，往往会得到意外的结果，因为不明显的隐式赋值会为对象创建新引用。控制台中的 `_` 变量是一例。调用跟踪对象也常导致意料之外的引用。

示例 **8-17**　弱引用是可调用的对象，返回的是被引用的对象；如果所指对象不存在了，返回 `None`

```
>>> import weakref
>>> a_set = {0, 1}
>>> wref = weakref.ref(a_set)  ❶
>>> wref
<weakref at 0x100637598; to 'set' at 0x100636748>
>>> wref()  ❷
{0, 1}
>>> a_set = {2, 3, 4}  ❸
>>> wref()  ❹
{0, 1}
>>> wref() is None  ❺
```

```
False
>>> wref() is None  ❻
True
```

❶ 创建弱引用对象 wref，下一行审查它。

❷ 调用 wref() 返回的是被引用的对象，{0, 1}。因为这是控制台会话，所以 {0, 1} 会绑定给 _ 变量。

❸ a_set 不再指代 {0, 1} 集合，因此集合的引用数量减少了。但是 _ 变量仍然指代它。

❹ 调用 wref() 依旧返回 {0, 1}。

❺ 计算这个表达式时，{0, 1} 存在，因此 wref() 不是 None。但是，随后 _ 绑定到结果值 False。现在 {0, 1} 没有强引用了。

❻ 因为 {0, 1} 对象不存在了，所以 wref() 返回 None。

weakref 模块的文档（http://docs.python.org/3/library/weakref.html）指出，weakref.ref 类其实是低层接口，供高级用途使用，多数程序最好使用 weakref 集合和 finalize。也就是说，应该使用 WeakKeyDictionary、WeakValueDictionary、WeakSet 和 finalize（在内部使用弱引用），不要自己动手创建并处理 weakref.ref 实例。我们在示例 8-17 中那么做是希望借助实际使用 weakref.ref 来褪去它的神秘色彩。但是实际上，多数时候 Python 程序都使用 weakref 集合。

下一节简要讨论 weakref 集合。

8.6.1 WeakValueDictionary简介

WeakValueDictionary 类实现的是一种可变映射，里面的值是对象的弱引用。被引用的对象在程序中的其他地方被当作垃圾回收后，对应的键会自动从 WeakValueDictionary 中删除。因此，WeakValueDictionary 经常用于缓存。

我们对 `WeakValueDictionary` 的演示受到来自英国六人喜剧团体 Monty Python 的经典短剧《奶酪店》的启发，在那出短剧里，客户问了 40 多种奶酪，包括切达干酪和马苏里拉奶酪，但是都没有货。[3]

> [3]`cheeseshop.python.org` 还是 PyPI（Python Package Index 软件仓库）的别名，一开始里面什么也没有。写作本书时，Python Cheese Shop 中有 41 426 个包。还不错，但是与有 131 000 个模块的 CPAN（Comprehensive Perl Archive Network）相比，还差得远。所有动态语言社区都羡慕 CPAN 中有那么多模块。

示例 8-18 实现一个简单的类，表示各种奶酪。

示例 **8-18**　`Cheese` 有个 `kind` 属性和标准的字符串表示形式

```
class Cheese:

def __init__(self, kind):
    self.kind = kind

def __repr__(self):
    return 'Cheese(%r)' % self.kind
```

在示例 8-19 中，我们把 `catalog` 中的各种奶酪载入 `WeakValueDictionary` 实现的 `stock` 中。然而，删除 `catalog` 后，`stock` 中只剩下一种奶酪了。你知道为什么帕尔马干酪（Parmesan）比其他奶酪保存的时间长吗？[4] 代码后面的提示中有答案。

> [4]帕尔马干酪在工厂中至少要存储一年，因此它比其他新鲜奶酪的保存时间长。但是，这不是我们想要的答案。

示例 **8-19**　顾客：“你们店里到底有没有奶酪？”

```
>>> import weakref
>>> stock = weakref.WeakValueDictionary()  ❶
>>> catalog = [Cheese('Red Leicester'), Cheese('Tilsit'),
...                 Cheese('Brie'), Cheese('Parmesan')]
...
>>> for cheese in catalog:
...     stock[cheese.kind] = cheese  ❷
...
```

```
>>> sorted(stock.keys())
['Brie', 'Parmesan', 'Red Leicester', 'Tilsit']  ❸
>>> del catalog
>>> sorted(stock.keys())
['Parmesan']  ❹
>>> del cheese
>>> sorted(stock.keys())
[]
```

❶ `stock` 是 `WeakValueDictionary` 实例。

❷ `stock` 把奶酪的名称映射到 `catalog` 中 `Cheese` 实例的弱引用上。

❸ `stock` 是完整的。

❹ 删除 `catalog` 之后，`stock` 中的大多数奶酪都不见了，这是 `WeakValueDictionary` 的预期行为。为什么不是全部呢？

> 临时变量引用了对象，这可能会导致该变量的存在时间比预期长。通常，这对局部变量来说不是问题，因为它们在函数返回时会被销毁。但是在示例 8-19 中，`for` 循环中的变量 `cheese` 是全局变量，除非显式删除，否则不会消失。

与 `WeakValueDictionary` 对应的是 `WeakKeyDictionary`，后者的键是弱引用。`weakref.WeakKeyDictionary` 的文档（https://docs.python.org/3/library/weakref.html?highlight=weakref#weakref.WeakKeyDictionary）指出了一些可能的用途：

> （`WeakKeyDictionary` 实例）可以为应用中其他部分拥有的对象附加数据，这样就无需为对象添加属性。这对覆盖属性访问权限的对象尤其有用。

`weakref` 模块还提供了 `WeakSet` 类，按照文档的说明，这个类的作用很简单："保存元素弱引用的集合类。元素没有强引用时，集合会把它删除。"如果一个类需要知道所有实例，一种好的方案是创建一个 `WeakSet` 类型的类属性，保存实例的引用。如果使用常规的 `set`，实例永远不会被垃圾回收，因为类中有实例的强引用，而类存在的时间与 Python 进程一样长，除非显式删除类。

这些集合，以及一般的弱引用，能处理的对象类型有限。参见下一节的说明。

8.6.2 弱引用的局限

不是每个 Python 对象都可以作为弱引用的目标（或称所指对象）。基本的 `list` 和 `dict` 实例不能作为所指对象，但是它们的子类可以轻松地解决这个问题：

```
class MyList(list):
    """list的子类，实例可以作为弱引用的目标"""

a_list = MyList(range(10))

# a_list可以作为弱引用的目标
wref_to_a_list = weakref.ref(a_list)
```

`set` 实例可以作为所指对象，因此实例 8-17 才使用 `set` 实例。用户定义的类型也没问题，这就解释了示例 8-19 中为什么使用那个简单的 `Cheese` 类。但是，`int` 和 `tuple` 实例不能作为弱引用的目标，甚至它们的子类也不行。

这些局限基本上是 CPython 的实现细节，在其他 Python 解释器中情况可能不一样。这些局限是内部优化导致的结果，下一节会以其中几个类型为例讨论（完全选读）。

8.7 Python对不可变类型施加的把戏

> 你可以放心跳过本节。这里讨论的是 Python 的实现细节，对 Python 用户来说没那么重要。这些细节是 CPython 核心开发者走的捷径和做的优化措施，对这门语言的用户而言无需了解，而且那些细节对其他 Python 实现可能没用，CPython 未来的版本可能也不会用。尽管如此，在学习别名和副本的过程中，你可能偶然见过这些把戏，因此我觉得有必要讲一下。

我惊讶地发现，对元组 `t` 来说，`t[:]` 不创建副本，而是返回同一个对象的引用。此外，`tuple(t)` 获得的也是同一个元组的引用。[5] 示例 8-20 证明了这一点。

> [5]文档明确指出了这个行为。在 Python 控制台中输入 `help(tuple)`，你会看到这句话："如果参数是一个元组，那么返回值是同一个对象。"撰写这本书之前，我还以为自己对元组无所不知。

示例 **8-20**　使用另一个元组构建元组，得到的其实是同一个元组

```
>>> t1 = (1, 2, 3)
>>> t2 = tuple(t1)
>>> t2 is t1  ❶
True
>>> t3 = t1[:]
>>> t3 is t1  ❷
True
```

❶ `t1` 和 `t2` 绑定到同一个对象。

❷ `t3` 也是。

`str`、`bytes` 和 `frozenset` 实例也有这种行为。注意，`frozenset` 实例不是序列，因此不能使用 `fs[:]`（`fs` 是一个 `frozenset` 实例）。但是，`fs.copy()` 具有相同的效果：它会欺骗你，返回同一个对象的引用，而不是创建一个副本，如示例 8-21 所示。[6]

[6]copy 方法不会复制所有对象，这是一个善意的谎言，为的是接口的兼容性：这使得 frozenset 的兼容性比 set 强。两个不可变对象是同一个对象还是副本，反正对最终用户来说没有区别。

示例 **8-21** 字符串字面量可能会创建共享的对象

```
>>> t1 = (1, 2, 3)
>>> t3 = (1, 2, 3)  # ❶
>>> t3 is t1  # ❷
False
>>> s1 = 'ABC'
>>> s2 = 'ABC'  # ❸
>>> s2 is s1  # ❹
True
```

❶ 新建一个元组。

❷ t1 和 t3 相等，但不是同一个对象。

❸ 再新建一个字符串。

❹ 奇怪的事发生了，a 和 b 指代同一个字符串。

共享字符串字面量是一种优化措施，称为驻留（interning）。CPython 还会在小的整数上使用这个优化措施，防止重复创建“热门”数字，如 0、-1 和 42。注意，CPython 不会驻留所有字符串和整数，驻留的条件是实现细节，而且没有文档说明。

千万不要依赖字符串或整数的驻留！比较字符串或整数是否相等时，应该使用 ==，而不是 is。驻留是 Python 解释器内部使用的一个特性。

本节讨论的把戏，包括 frozenset.copy() 的行为，是“善意的谎言”，能节省内存，提升解释器的速度。别担心，它们不会为你带来任何麻烦，因为只有不可变类型会受到影响。或许这些细枝末节的最佳用途是与其他 Python 程序员打赌，提高自己的胜算。

8.8　本章小结

每个 Python 对象都有标识、类型和值。只有对象的值会不时变化。[7]

> [7]其实，对象的类型也可以变，方法只有一种：为 `__class__` 属性指定其他类。但这是在作恶，我后悔加上这个脚注了。

如果两个变量指代的不可变对象具有相同的值（`a == b` 为 `True`），实际上它们指代的是副本还是同一个对象的别名基本没什么关系，因为不可变对象的值不会变，但有一个例外。这里说的例外是不可变的集合，如元组和 `frozenset`：如果不可变集合保存的是可变元素的引用，那么可变元素的值发生变化后，不可变集合也会随之改变。实际上，这种情况不是很常见。不可变集合不变的是所含对象的标识。

变量保存的是引用，这一点对 Python 编程有很多实际的影响。

- 简单的赋值不创建副本。
- 对 `+=` 或 `*=` 所做的增量赋值来说，如果左边的变量绑定的是不可变对象，会创建新对象；如果是可变对象，会就地修改。
- 为现有的变量赋予新值，不会修改之前绑定的变量。这叫重新绑定：现在变量绑定了其他对象。如果变量是之前那个对象的最后一个引用，对象会被当作垃圾回收。
- 函数的参数以别名的形式传递，这意味着，函数可能会修改通过参数传入的可变对象。这一行为无法避免，除非在本地创建副本，或者使用不可变对象（例如，传入元组，而不传入列表）。
- 使用可变类型作为函数参数的默认值有危险，因为如果就地修改了参数，默认值也就变了，这会影响以后使用默认值的调用。

在 CPython 中，对象的引用数量归零后，对象会被立即销毁。如果除了循环引用之外没有其他引用，两个对象都会被销毁。某些情况下，可能需要保存对象的引用，但不留存对象本身。例如，有一个类想要记录所有实例。这个需求可以使用弱引用实现，这是一种低层机制，是

weakref 模块中 WeakValueDictionary、WeakKeyDictionary 和 WeakSet 等有用的集合类，以及 finalize 函数的底层支持。

8.9　延伸阅读

Python 语言参考手册中“Data Model”一章（https://docs.python.org/3/reference/datamodel.html）的开头清楚解释了对象的标识和值。

“Python 核心系列”图书的作者 Wesley Chun 在 OSCON 2013 做了一场精彩的演讲，涵盖了本章讨论的很多话题。在“Python 103: Memory Model & Best Practices”演讲页面（http://conferences.oreilly.com/oscon/oscon2013/public/schedule/detail/2937
可以下载幻灯片。Wesley 在 EuroPython 2011 还做过一次更长的演讲（YouTube 视频：https://www.youtube.com/watch?v=HHFCFJSPWrI），不仅涵盖了本章的主题，还讨论了特殊方法的使用。

Doug Hellmann 写了一长串精彩的博客文章，题为“Python Module of the Week”（http://pymotw.com），[8] 后来集结成书，即《Python 标准库》。他写的“copy - Duplicate Objects”（http://pymotw.com/2/copy/）[9] 和“weakref - Garbage-Collectable References to Objects”（http://pymotw.com/2/weakref/）[10] 两篇文章涵盖了本章讨论的部分话题。

[8]原来是基于 Python 2 的（https://pymotw.com/2/），现在已经改为基于 Python 3（https://pymotw.com/3/）。——编者注

[9]新的版本基于 Python 3（https://pymotw.com/3/copy/）。——编者注

[10]新的版本基于 Python 3，并改名为“weakref - Impermanent References to Objects”（https://pymotw.com/3/weakref/）。——编者注

关于 CPython 分代垃圾回收程序的更多信息，请参阅 gc 模块的文档（https://docs.python.org/3/library/gc.html）。文档开头的第一句话是：“这个模块为可选的垃圾回收程序提供接口。”“可选的”这个修饰词可能让人惊讶，不过“Data Model”一章（https://docs.python.org/3/reference/datamodel.html）也说：

> 垃圾回收可以延缓实现，或者完全不实现——如何实现垃圾回收是

实现的质量问题，只要不把还能获得的对象给回收了就行。

Fredrik Lundh（很多核心库的创建者，如 ElementTree、Tkinter 和图像库 PIL）写了一篇短文，谈论了 Python 的垃圾回收程序，题为“How Does Python Manage Memory?”（http://effbot.org/pyfaq/how-does-python-manage-memory.htm）。他强调垃圾回收程序是一种实现的特性，其行为在不同的 Python 解释器中有所不同。例如，Jython 用的是 Java 垃圾回收程序。

CPython 3.4 改进了处理有 `__del__` 方法的对象的方式，参见“PEP 442—Safe object finalization”（https://www.python.org/dev/peps/pep-0442/）。

维基百科中有一篇文章讲解了字符串驻留（https://en.wikipedia.org/wiki/String_interning），那篇文章提到了几种语言对这个技术的利用，包括 Python。

杂谈

平等对待所有对象

发现 Python 之前，我学过Java。我一直觉得 Java 的 `==` 运算符用着不舒服。程序员关注的基本上是相等性，而不是标识，但是 Java 的 `==` 运算符比较的是对象（不是基本类型）的引用，而不是对象的值。就算是比较字符串这样的基本操作，Java 也强制你使用 `.equals` 方法。尽管如此，`.equals` 方法还有另一个问题：如果编写 `a.equals(b)`，而 `a` 是 `null`，会得到一个空指针异常。Java 设计者觉得有必要重载字符串的 `+` 运算符，那为什么不把 `==` 也重载了？

Python 采取了正确的方式。`==` 运算符比较对象的值，而 `is` 比较引用。此外，Python 支持重载运算符，`==` 能正确处理标准库中的所有对象，包括 `None`——这是一个正常的对象，与 Java 的 `null` 不同。

当然，你可以在自己的类中定义 `__eq__` 方法，决定 `==` 如何比较实例。如果不覆盖 `__eq__` 方法，那么从 `object` 继承的方法比较对象的 ID，因此这种后备机制认为用户定义的类的各个实例是不同的。

1998 年 9 月的一个下午，读完 Python 教程后，考虑到这些行为，我立即就从 Java 转到 Python 了。

可变性

如果所有 Python 对象都是不可变的，那么本章就没有存在的必要了。处理不可变的对象时，变量保存的是真正的对象还是共享对象的引用无关紧要。如果 `a == b` 成立，而且两个对象都不会变，那么它们就可能是相同的对象。这就是为什么字符串可以安全使用驻留。仅当对象可变时，对象标识才重要。

在“纯”函数式编程中，所有数据都是不可变的，如果为集合追加元素，那么其实会创建新的集合。然而，Python 不是函数式语言，更别提纯不纯了。在 Python 中，用户定义的类，其实例默认可变（多数面向对象语言都是如此）。自己创建对象时，如果需要不可变的对象，一定要格外小心。此时，对象的每个属性都必须是不可变的，否则会出现类似元组那种行为：元组本身不可变，但是如果里面保存着可变对象，那么元组的值可能会变。

可变对象还是导致多线程编程难以处理的主要原因，因为某个线程改动对象后，如果不正确地同步，那就会损坏数据。但是过度同步又会导致死锁。

对象析构和垃圾回收

Python 没有直接销毁对象的机制，这一疏漏其实是一个好的特性：如果随时可以销毁对象，那么指向对象的强引用怎么办？

CPython 中的垃圾回收主要依靠引用计数，这容易实现，但是遇到引用循环容易泄露内存，因此 CPython 2.0（2000 年 10 月发布）实现了分代垃圾回收程序，它能把引用循环中不可获取的对象销毁。

但是引用计数仍然作为一种基准存在，一旦引用数量归零，就立即销毁对象。这意味着，在 CPython 中，这样写是安全的（至少目前如此）：

```
open('test.txt', 'wt', encoding='utf-8').write('1, 2, 3')
```

这行代码是安全的，因为文件对象的引用数量会在 `write` 方法返回后归零，Python 在销毁内存中表示文件的对象之前，会立即关闭文件。然而，这行代码在 Jython 或 IronPython 中却不安全，因为它们使用的是宿主运行时（Java VM 和 .NET CLR）中的垃圾回收程序，那些回收程序更复杂，但是不依靠引用计数，而且销毁对象和关闭文件的时间可能更长。在任何情况下，包括 CPython，最好显式关闭文件；而关闭文件的最可靠方式是使用 `with` 语句，它能保证文件一定会被关闭，即使打开文件时抛出了异常也无妨。使用 `with`，上述代码片段变成了：

```
with open('test.txt', 'wt', encoding='utf-8') as fp:
    fp.write('1, 2, 3')
```

如果对垃圾回收程序感兴趣，可以阅读 Thomas Perl 的论文，“Python Garbage Collector Implementations: CPython, PyPy and GaS”（https://thp.io/2012/python-gc/python_gc_final_2012-01-22.pdf）。就是从那篇论文中，我得知在 CPython 中 `open().write()` 是安全的。

参数传递：共享传参

解释 Python 中参数传递的方式时，人们经常这样说：“参数按值传递，但是这里的值是引用。”这么说没错，但是会引起误解，因为在旧式语言中，最常用的参数传递模式有按值传递（函数得到参数的副本）和按引用传递（函数得到参数的指针）。在 Python 中，函数得到参数的副本，但是参数始终是引用。因此，如果参数引用的是可变对象，那么对象可能会被修改，但是对象的标识不变。此外，因为函数得到的是参数引用的副本，所以重新绑定对函数外部没有影响。读过《程序设计语言——实践之路（第 3 版）》[11]（Michael L. Scott 著）之后，尤其是 8.3.1 节“参数模式”，我决定采用共享传参（call by sharing）这个说法。

爱丽丝和白骑士关于那首歌的对话完整版

我喜欢这段对话，但是放在一章的开头太长了。下面是关于白骑士那首歌的完整对话，谈到了曲名和得名的缘由。

“你不开心，”白骑士用一种忧虑的声调说，“让我给你唱一首歌安慰你吧。”

“那首歌很长吗？”爱丽丝问道，因为这一天她已经听过许多诗了。

“是很长，”白骑士说，“不过它非常、非常美。不论谁听到我唱这首歌——或者是听得热泪盈眶，或者是——”

“或者是什么呀？”爱丽丝问道，因为白骑士忽然煞住不言语了。

“或者是没有热泪盈眶，你知道。这首歌的曲名叫作：《黑线鳕的眼睛》。”

“哦，那是一首歌的曲名，是吗？”爱丽丝问道，她试着使自己感到有兴趣。

“不，你不明白，”白骑士说，看来有些心烦的样子,“那是人家这么叫的曲名。

真正的曲名是《老而又老的老头儿》。”

“那么我刚才应该说，‘那首歌是那么被人叫的’？”爱丽丝自己纠正说。

“不，你不应该这么说。这是另一码事！这首歌人家叫作《方法和手段》。不过这只不过是人家这样叫，你知道！”

“嗯，那么，那究竟是什么歌呢？”爱丽丝问道，她这一次完完全全给弄糊涂了。

“我正是准备说的呀，”白骑士说道，“这首歌真正是《坐在大门上》；曲子是我自己发明的。”

——Lewis Carroll
《爱丽丝镜中奇遇记》，第 8 章“这是我自己的发明”

[11]该书英文版（书名：*Programming Language Pragmatics*）在 2015 年 12 月已出第 4 版。——编者注

第 9 章　符合Python风格的对象

> 绝对不要使用两个前导下划线，这是很烦人的自私行为。[1]
>
> ——Ian Bicking
> pip、virtualenv 和 Paste 等项目的创建者

[1]摘自 Paste 的风格指南（http://pythonpaste.org/StyleGuide.html）。

得益于 Python 数据模型，自定义类型的行为可以像内置类型那样自然。实现如此自然的行为，靠的不是继承，而是鸭子类型（duck typing）：我们只需按照预定行为实现对象所需的方法即可。

前一章分析了很多内置对象的结构和行为，这一章则自己定义类，而且让类的行为跟真正的 Python 对象一样。

这一章接续第 1 章，说明如何实现在很多 Python 类型中常见的特殊方法。

本章包含以下话题：

- 支持用于生成对象其他表示形式的内置函数（如 `repr()`、`bytes()`，等等）
- 使用一个类方法实现备选构造方法
- 扩展内置的 `format()` 函数和 `str.format()` 方法使用的格式微语言
- 实现只读属性
- 把对象变为可散列的，以便在集合中及作为 `dict` 的键使用
- 利用 `__slots__` 节省内存

我们将开发一个简单的二维欧几里得向量类型，在这个过程中涵盖上述

全部话题。

在实现这个类型的中间阶段，我们会讨论两个概念：

- 如何以及何时使用 `@classmethod` 和 `@staticmethod` 装饰器
- Python 的私有属性和受保护属性的用法、约定和局限

我们从对象表示形式函数开始。

9.1 对象表示形式

每门面向对象的语言至少都有一种获取对象的字符串表示形式的标准方式。Python 提供了两种方式。

`repr()`

　　以便于开发者理解的方式返回对象的字符串表示形式。

`str()`

　　以便于用户理解的方式返回对象的字符串表示形式。

正如你所知，我们要实现 `__repr__` 和 `__str__` 特殊方法，为 `repr()` 和 `str()` 提供支持。

为了给对象提供其他的表示形式，还会用到另外两个特殊方法：`__bytes__` 和 `__format__`。`__bytes__` 方法与 `__str__` 方法类似：`bytes()` 函数调用它获取对象的字节序列表示形式。而 `__format__` 方法会被内置的 `format()` 函数和 `str.format()` 方法调用，使用特殊的格式代码显示对象的字符串表示形式。我们将在下一个示例中讨论 `__bytes__` 方法，随后再讨论 `__format__` 方法。

> 如果你是从 Python 2 转过来的，记住，在 Python 3 中，`__repr__`、`__str__` 和 `__format__` 都必须返回 Unicode 字符串（`str` 类型）。只有 `__bytes__` 方法应该返回字节序列（`bytes` 类型）。

9.2　再谈向量类

为了说明用于生成对象表示形式的众多方法，我们将使用一个 `Vector2d` 类，它与第 1 章中的类似。这一节和接下来的几节会不断实现这个类。我们期望 `Vector2d` 实例具有的基本行为如示例 9-1 所示。

示例 **9-1**　`Vector2d` 实例有多种表示形式

```
>>> v1 = Vector2d(3, 4)
>>> print(v1.x, v1.y)  ❶
3.0 4.0
>>> x, y = v1  ❷
>>> x, y
(3.0, 4.0)
>>> v1  ❸
Vector2d(3.0, 4.0)
>>> v1_clone = eval(repr(v1))  ❹
>>> v1 == v1_clone  ❺
True
>>> print(v1)  ❻
(3.0, 4.0)
>>> octets = bytes(v1)  ❼
>>> octets
b'd\\x00\\x00\\x00\\x00\\x00\\x00\\x08@\\x00\\x00\\x00\\x00\\x00\\x00\\x10@'
>>> abs(v1)  ❽
5.0
>>> bool(v1), bool(Vector2d(0, 0))  ❾
```

❶ `Vector2d` 实例的分量可以直接通过属性访问（无需调用读值方法）。

❷ `Vector2d` 实例可以拆包成变量元组。

❸ `repr` 函数调用 `Vector2d` 实例，得到的结果类似于构建实例的源码。

❹ 这里使用 `eval` 函数，表明 `repr` 函数调用 `Vector2d` 实例得到的是对构造方法的准确表述。[2]

[2]这里使用 eval 函数克隆对象是为了说明 repr 方法。使用 copy.copy 函数克隆实例更安全也更快速。

❺ Vector2d 实例支持使用 == 比较；这样便于测试。

❻ print 函数会调用 str 函数，对 Vector2d 来说，输出的是一个有序对。

❼ bytes 函数会调用 __bytes__ 方法，生成实例的二进制表示形式。

❽ abs 函数会调用 __abs__ 方法，返回 Vector2d 实例的模。

❾ bool 函数会调用 __bool__ 方法，如果 Vector2d 实例的模为零，返回 False，否则返回 True。

示例 9-1 中的 Vector2d 类在 vector2d_v0.py 文件中实现（见示例 9-2）。这段代码基于示例 1-2，除了 == 之外（在测试中用得到），其他中缀运算符将在第 13 章实现。现在，Vector2d 用到了几个特殊方法，这些方法提供的操作是 Python 高手期待设计良好的对象所提供的。

示例 **9-2**　vector2d_v0.py：目前定义的都是特殊方法

```
from array import array
import math


class Vector2d:
    typecode = 'd'  ❶

    def __init__(self, x, y):
        self.x = float(x)   ❷
        self.y = float(y)

    def __iter__(self):
        return (i for i in (self.x, self.y))  ❸

    def __repr__(self):
        class_name = type(self).__name__
        return '{}({!r}, {!r})'.format(class_name, *self)  ❹

    def __str__(self):
        return str(tuple(self))  ❺
```

```
    def __bytes__(self):
        return (bytes([ord(self.typecode)]) +  ❻
                bytes(array(self.typecode, self)))  ❼

    def __eq__(self, other):
        return tuple(self) == tuple(other)  ❽

    def __abs__(self):
        return math.hypot(self.x, self.y)  ❾

    def __bool__(self):
        return bool(abs(self))  ❿
```

❶ `typecode` 是类属性，在 `Vector2d` 实例和字节序列之间转换时使用。

❷ 在 `__init__` 方法中把 `x` 和 `y` 转换成浮点数，尽早捕获错误，以防调用 `Vector2d` 函数时传入不当参数。

❸ 定义 `__iter__` 方法，把 `Vector2d` 实例变成可迭代的对象，这样才能拆包（例如，`x, y = my_vector`）。这个方法的实现方式很简单，直接调用生成器表达式一个接一个产出分量。[3]

> [3]这一行也可以写成 `yield self.x; yield.self.y`。第 14 章会进一步讨论 `__iter__` 特殊方法、生成器表达式和 `yield` 关键字。

❹ `__repr__` 方法使用 `{!r}` 获取各个分量的表示形式，然后插值，构成一个字符串；因为 `Vector2d` 实例是可迭代的对象，所以 `*self` 会把 `x` 和 `y` 分量提供给 `format` 函数。

❺ 从可迭代的 `Vector2d` 实例中可以轻松地得到一个元组，显示为一个有序对。

❻ 为了生成字节序列，我们把 `typecode` 转换成字节序列，然后……

❼ ……迭代 `Vector2d` 实例，得到一个数组，再把数组转换成字节序列。

❽ 为了快速比较所有分量，在操作数中构建元组。对 `Vector2d` 实例来说，可以这样做，不过仍有问题。参见下面的警告。

❾ 模是 `x` 和 `y` 分量构成的直角三角形的斜边长。

❿ `__bool__` 方法使用 `abs(self)` 计算模，然后把结果转换成布尔值，因此，`0.0` 是 `False`，非零值是 `True`。

> 示例 9-2 中的 `__eq__` 方法，在两个操作数都是 `Vector2d` 实例时可用，不过拿 `Vector2d` 实例与其他具有相同数值的可迭代对象相比，结果也是 `True`（如 `Vector(3, 4) == [3, 4]`）。这个行为可以视作特性，也可以视作缺陷。第 13 章讲到运算符重载时才能进一步讨论。

我们已经定义了很多基本方法，但是显然少了一个操作：使用 `bytes()` 函数生成的二进制表示形式重建 `Vector2d` 实例。

9.3　备选构造方法

我们可以把 `Vector2d` 实例转换成字节序列了；同理，也应该能从字节序列转换成 `Vector2d` 实例。在标准库中探索一番之后，我们发现 `array.array` 有个类方法 `.frombytes`（2.9.1 节介绍过）正好符合需求。下面在 vector2d_v1.py（见示例 9-3）中为 `Vector2d` 定义一个同名类方法。

示例 **9-3**　vector2d_v1.py 的一部分：这段代码只列出了 `frombytes` 类方法，要添加到 vector2d_v0.py（见示例 9-2）中定义的`Vector2d` 类中

```
    @classmethod  ❶
    def frombytes(cls, octets):  ❷
        typecode = chr(octets[0])  ❸
        memv = memoryview(octets[1:]).cast(typecode)  ❹
        return cls(*memv)  ❺
```

❶ 类方法使用 `classmethod` 装饰器修饰。

❷ 不用传入 `self` 参数；相反，要通过 `cls` 传入类本身。

❸ 从第一个字节中读取 `typecode`。

❹ 使用传入的 `octets` 字节序列创建一个 `memoryview`，然后使用 `typecode` 转换。[4]

> [4]2.9.2 节简单介绍过 `memoryview`，说明了它的 `.cast` 方法。

❺ 拆包转换后的 `memoryview`，得到构造方法所需的一对参数。

我们用的 `classmethod` 装饰器是 Python 专用的，下面讲解一下。

9.4　classmethod与staticmethod

Python 教程没有提到 classmethod 装饰器，也没有提到 staticmethod。学过 Java 面向对象编程的人可能觉得奇怪，为什么 Python 提供两个这样的装饰器，而不是只提供一个？

先来看 classmethod。示例 9-3 展示了它的用法：定义操作类，而不是操作实例的方法。classmethod 改变了调用方法的方式，因此类方法的第一个参数是类本身，而不是实例。classmethod 最常见的用途是定义备选构造方法，例如示例 9-3 中的 frombytes。注意，frombytes 的最后一行使用 cls 参数构建了一个新实例，即 cls(*memv)。按照约定，类方法的第一个参数名为 cls（但是 Python 不介意具体怎么命名）。

staticmethod 装饰器也会改变方法的调用方式，但是第一个参数不是特殊的值。其实，静态方法就是普通的函数，只是碰巧在类的定义体中，而不是在模块层定义。示例 9-4 对 classmethod 和 staticmethod 的行为做了对比。

示例 **9-4**　比较 classmethod 和 staticmethod 的行为

```
>>> class Demo:
...     @classmethod
...     def klassmeth(*args):
...         return args  # ❶
...     @staticmethod
...     def statmeth(*args):
...         return args  # ❷
...
>>> Demo.klassmeth()  # ❸
(<class '__main__.Demo'>,)
>>> Demo.klassmeth('spam')
(<class '__main__.Demo'>, 'spam')
>>> Demo.statmeth()   # ❹
()
>>> Demo.statmeth('spam')
('spam',)
```

❶ `klassmeth` 返回全部位置参数。

❷ `statmeth` 也是。

❸ 不管怎样调用 `Demo.klassmeth`，它的第一个参数始终是 `Demo` 类。

❹ `Demo.statmeth` 的行为与普通的函数相似。

> `classmethod` 装饰器非常有用，但是我从未见过不得不用 `staticmethod` 的情况。如果想定义不需要与类交互的函数，那么在模块中定义就好了。有时，函数虽然从不处理类，但是函数的功能与类紧密相关，因此想把它放在近处。即便如此，在同一模块中的类前面或后面定义函数也就行了。[5]

[5]本书的技术审校之一 Leonardo Rochael 不同意我对 `staticmethod` 的见解，作为反驳，他推荐阅读 Julien Danjou 写的一篇博客文章，题为"The Definitive Guide on How to Use Static, Class or Abstract Methods in Python"（https://julien.danjou.info/blog/2013/guide-python-static-class-abstract-methods）。Danjou 的这篇文章写得很好，我推荐阅读。但是，我对 `staticmethod` 的观点依然不变。请读者自辨。

现在，我们对 `classmethod` 的作用已经有所了解（而且知道 `staticmethod` 不是特别有用），下面继续讨论对象的表示形式，说明如何支持格式化输出。

9.5 格式化显示

内置的 format() 函数和 str.format() 方法把各个类型的格式化方式委托给相应的 .__format__(format_spec) 方法。format_spec 是格式说明符，它是：

- format(my_obj, format_spec) 的第二个参数，或者
- str.format() 方法的格式字符串，{} 里代换字段中冒号后面的部分

例如：

```
>>> brl = 1/2.43  # BRL到USD的货币兑换比价
>>> brl
0.4115226337448559
>>> format(brl, '0.4f')  # ❶
'0.4115'
>>> '1 BRL = {rate:0.2f} USD'.format(rate=brl)  # ❷
'1 BRL = 0.41 USD'
```

❶ 格式说明符是 '0.4f'。

❷ 格式说明符是 '0.2f'。代换字段中的 'rate' 子串是字段名称，与格式说明符无关，但是它决定把 .format() 的哪个参数传给代换字段。

第 2 条标注指出了一个重要知识点：'{0.mass:5.3e}' 这样的格式字符串其实包含两部分，冒号左边的 '0.mass' 在代换字段句法中是字段名，冒号后面的 '5.3e' 是格式说明符。格式说明符使用的表示法叫格式规范微语言（“Format Specification Mini-Language”，https://docs.python.org/3/library/string.html#formatspec）。

如果你对 format() 和 str.format() 都感到陌生，根据我的教学经验，最好先学 format() 函数，因为它只使用格式规范微语言。学会这些表示法之后，再阅读格式字符串句法（“Format

String Syntax"，https://docs.python.org/3/library/string.html#formatspec），学习 str.format() 方法使用的 {:} 代换字段表示法（包含转换标志 !s、!r 和 !a）。

格式规范微语言为一些内置类型提供了专用的表示代码。比如，b 和 x 分别表示二进制和十六进制的 int 类型，f 表示小数形式的 float 类型，而 % 表示百分数形式：

```
>>> format(42, 'b')
'101010'
>>> format(2/3, '.1%')
'66.7%'
```

格式规范微语言是可扩展的，因为各个类可以自行决定如何解释 format_spec 参数。例如， datetime 模块中的类，它们的 __format__ 方法使用的格式代码与 strftime() 函数一样。下面是内置的 format() 函数和 str.format() 方法的几个示例：

```
>>> from datetime import datetime
>>> now = datetime.now()
>>> format(now, '%H:%M:%S')
'18:49:05'
>>> "It's now {:%I:%M %p}".format(now)
"It's now 06:49 PM"
```

如果类没有定义 __format__ 方法，从 object 继承的方法会返回 str(my_object)。我们为 Vector2d 类定义了 __str__ 方法，因此可以这样做：

```
>>> v1 = Vector2d(3, 4)
>>> format(v1)
'(3.0, 4.0)'
```

然而，如果传入格式说明符，object.__format__ 方法会抛出 TypeError：

```
>>> format(v1, '.3f')
Traceback (most recent call last):
```

```
  ...
TypeError: non-empty format string passed to object.__format__
```

我们将实现自己的微语言来解决这个问题。首先，假设用户提供的格式说明符是用于格式化向量中各个浮点数分量的。我们想达到的效果是：

```
>>> v1 = Vector2d(3, 4)
>>> format(v1)
'(3.0, 4.0)'
>>> format(v1, '.2f')
'(3.00, 4.00)'
>>> format(v1, '.3e')
'(3.000e+00, 4.000e+00)'
```

实现这种输出的 `__format__` 方法如示例 9-5 所示。

示例 **9-5** `Vector2d.__format__` 方法，第 1 版

```
    # 在Vector2d类中定义

    def __format__(self, fmt_spec=''):
        components = (format(c, fmt_spec) for c in self)  # ❶
        return '({}, {})'.format(*components)  # ❷
```

❶ 使用内置的 `format` 函数把 `fmt_spec` 应用到向量的各个分量上，构建一个可迭代的格式化字符串。

❷ 把格式化字符串代入公式 `'(x, y)'` 中。

下面要在微语言中添加一个自定义的格式代码：如果格式说明符以 `'p'` 结尾，那么在极坐标中显示向量，即 `<r, θ >`，其中 `r` 是模，`θ`（西塔）是弧度；其他部分（`'p'` 之前的部分）像往常那样解释。

为自定义的格式代码选择字母时，我会避免使用其他类型用过的字母。在格式规范微语言（https://docs.python.org/3/library/string.html#formatspec）中我们看到，整数使用的代码有 `'bcdoxXn'`，浮点数使用的代码有 `'eEfFgGn%'`，字符串使用的代码有 `'s'`。因此，我为极坐标选的

代码是 'p'。各个类使用自己的方式解释格式代码，在自定义的格式代码中重复使用代码字母不会出错，但是可能会让用户困惑。

对极坐标来说，我们已经定义了计算模的 __abs__ 方法，因此还要定义一个简单的 angle 方法，使用 math.atan2() 函数计算角度。angle 方法的代码如下：

```
# 在Vector2d类中定义

def angle(self):
    return math.atan2(self.y, self.x)
```

这样便可以增强 __format__ 方法，计算极坐标，如示例 9-6 所示。

示例 **9-6** Vector2d.__format__ 方法，第 2 版，现在能计算极坐标了

```
def __format__(self, fmt_spec=''):
    if fmt_spec.endswith('p'):  ❶
        fmt_spec = fmt_spec[:-1]  ❷
        coords = (abs(self), self.angle())  ❸
        outer_fmt = '<{}, {}>'  ❹
    else:
        coords = self  ❺
        outer_fmt = '({}, {})'  ❻
    components = (format(c, fmt_spec) for c in coords)  ❼
    return outer_fmt.format(*components)  ❽
```

❶ 如果格式代码以 'p' 结尾，使用极坐标。

❷ 从 fmt_spec 中删除 'p' 后缀。

❸ 构建一个元组，表示极坐标：(magnitude, angle)。

❹ 把外层格式设为一对尖括号。

❺ 如果不以 'p' 结尾，使用 self 的 x 和 y 分量构建直角坐标。

❻ 把外层格式设为一对圆括号。

❼ 使用各个分量生成可迭代的对象，构成格式化字符串。

❽ 把格式化字符串代入外层格式。

示例 9-6 中的代码得到的结果如下：

```
>>> format(Vector2d(1, 1), 'p')
'<1.4142135623730951, 0.7853981633974483>'
>>> format(Vector2d(1, 1), '.3ep')
'<1.414e+00, 7.854e-01>'
>>> format(Vector2d(1, 1), '0.5fp')
'<1.41421, 0.78540>'
```

如本节所示，为用户自定义的类型扩展格式规范微语言并不难。

下面换个话题，它不仅事关对象的外观：我们将把 `Vector2d` 变成可散列的，这样便可以构建向量集合，或者把向量当作 `dict` 的键使用。不过在此之前，必须让向量不可变。详情参见下一节。

9.6　可散列的Vector2d

按照定义，目前 Vector2d 实例是不可散列的，因此不能放入集合（set）中：

```
>>> v1 = Vector2d(3, 4)
>>> hash(v1)
Traceback (most recent call last):
  ...
TypeError: unhashable type: 'Vector2d'
>>> set([v1])
Traceback (most recent call last):
  ...
TypeError: unhashable type: 'Vector2d'
```

为了把 Vector2d 实例变成可散列的，必须使用 __hash__ 方法（还需要 __eq__ 方法，前面已经实现了）。此外，还要让向量不可变，详情参见第 3 章的附注栏“什么是可散列的数据类型”。

目前，我们可以为分量赋新值，如 v1.x = 7，Vector2d 类的代码并不阻止这么做。我们想要的行为是这样的：

```
>>> v1.x, v1.y
(3.0, 4.0)
>>> v1.x = 7
Traceback (most recent call last):
  ...
AttributeError: can't set attribute
```

为此，我们要把 x 和 y 分量设为只读特性，如示例 9-7 所示。

示例 **9-7**　vector2d_v3.py：这里只给出了让 Vector2d 不可变的代码，完整的代码清单在示例 9-9 中

```
class Vector2d:
    typecode = 'd'

    def __init__(self, x, y):
        self.__x = float(x)  ❶
```

```
        self.__y = float(y)

    @property  ❷
    def x(self):  ❸
        return self.__x  ❹

    @property  ❺
    def y(self):
        return self.__y

    def __iter__(self):
        return (i for i in (self.x, self.y))  ❻

    # 下面是其他方法（排版需要，省略了）
```

❶ 使用两个前导下划线（尾部没有下划线，或者有一个下划线），把属性标记为私有的。[6]

[6]根据本章开头引用的那句话，这不符合 Ian Bicking 的建议。私有属性的优缺点参见后面的 9.7 节。

❷ `@property` 装饰器把读值方法标记为特性。

❸ 读值方法与公开属性同名，都是 `x`。

❹ 直接返回 `self.__x`。

❺ 以同样的方式处理 `y` 特性。

❻ 需要读取 `x` 和 `y` 分量的方法可以保持不变，通过 `self.x` 和 `self.y` 读取公开特性，而不必读取私有属性，因此上述代码清单省略了这个类的其他代码。

> `Vector.x` 和 `Vector.y` 是只读特性。读写特性在第 19 章讨论，届时会深入说明 `@property` 装饰器。

注意，我们让这些向量不可变是有原因的，因为这样才能实现 `__hash__` 方法。这个方法应该返回一个整数，理想情况下还要考虑对象属性的散列值（`__eq__` 方法也要使用），因为相等的对象应该具有

相同的散列值。根据特殊方法 `__hash__` 的文档（https://docs.python.org/3/reference/datamodel.html），最好使用位运算符异或（^）混合各分量的散列值——我们会这么做。`Vector2d.__hash__` 方法的代码十分简单，如示例 9-8 所示。

示例 **9-8**　vector2d_v3.py：实现 `__hash__` 方法

```
# 在Vector2d类中定义

def __hash__(self):
    return hash(self.x) ^ hash(self.y)
```

添加 `__hash__` 方法之后，向量变成可散列的了：

```
>>> v1 = Vector2d(3, 4)
>>> v2 = Vector2d(3.1, 4.2)
>>> hash(v1), hash(v2)
(7, 384307168202284039)
>>> set([v1, v2])
{Vector2d(3.1, 4.2), Vector2d(3.0, 4.0)}
```

要想创建可散列的类型，不一定要实现特性，也不一定要保护实例属性。只需正确地实现 `__hash__` 和 `__eq__` 方法即可。但是，实例的散列值绝不应该变化，因此我们借机提到了只读特性。

如果定义的类型有标量数值，可能还要实现 `__int__` 和 `__float__` 方法（分别被 `int()` 和 `float()` 构造函数调用），以便在某些情况下用于强制转换类型。此外，还有用于支持内置的 `complex()` 构造函数的 `__complex__` 方法。`Vector2d` 或许应该提供 `__complex__` 方法，不过我把它留作练习给读者。

我们一直在定义 `Vector2d` 类，也列出了很多代码片段，示例 9-9 是整理后的完整代码清单，保存在 vector2d_v3.py 文件中，包含开发时我编写的全部 doctest。

示例 **9-9**　vector2d_v3.py：完整版

```
"""
A two-dimensional vector class

    >>> v1 = Vector2d(3, 4)
    >>> print(v1.x, v1.y)
    3.0 4.0
    >>> x, y = v1
    >>> x, y
    (3.0, 4.0)
    >>> v1
    Vector2d(3.0, 4.0)
    >>> v1_clone = eval(repr(v1))
    >>> v1 == v1_clone
    True
    >>> print(v1)
    (3.0, 4.0)
    >>> octets = bytes(v1)
    >>> octets
    b'd\\x00\\x00\\x00\\x00\\x00\\x00\\x08@\\x00\\x00\\x00\\x00\\x00\\x00\\x1
    >>> abs(v1)
    5.0
    >>> bool(v1), bool(Vector2d(0, 0))
    (True, False)


Test of ``.frombytes()`` class method:

    >>> v1_clone = Vector2d.frombytes(bytes(v1))
    >>> v1_clone
    Vector2d(3.0, 4.0)
    >>> v1 == v1_clone
    True


Tests of ``format()`` with Cartesian coordinates:

    >>> format(v1)
    '(3.0, 4.0)'
    >>> format(v1, '.2f')
    '(3.00, 4.00)'
    >>> format(v1, '.3e')
    '(3.000e+00, 4.000e+00)'


Tests of the ``angle`` method::

    >>> Vector2d(0, 0).angle()
```

```
    0.0
    >>> Vector2d(1, 0).angle()
    0.0
    >>> epsilon = 10**-8
    >>> abs(Vector2d(0, 1).angle() - math.pi/2) < epsilon
    True
    >>> abs(Vector2d(1, 1).angle() - math.pi/4) < epsilon
    True


Tests of ``format()`` with polar coordinates:

    >>> format(Vector2d(1, 1), 'p')  # doctest:+ELLIPSIS
    '<1.414213..., 0.785398...>'
    >>> format(Vector2d(1, 1), '.3ep')
    '<1.414e+00, 7.854e-01>'
    >>> format(Vector2d(1, 1), '0.5fp')
    '<1.41421, 0.78540>'


Tests of `x` and `y` read-only properties:

    >>> v1.x, v1.y
    (3.0, 4.0)
    >>> v1.x = 123
    Traceback (most recent call last):
      ...
    AttributeError: can't set attribute


Tests of hashing:

    >>> v1 = Vector2d(3, 4)
    >>> v2 = Vector2d(3.1, 4.2)
    >>> hash(v1), hash(v2)
    (7, 384307168202284039)
    >>> len(set([v1, v2]))
    2

"""

from array import array
import math

class Vector2d:
    typecode = 'd'

    def __init__(self, x, y):
```

```
        self.__x = float(x)
        self.__y = float(y)

    @property
    def x(self):
        return self.__x

    @property
    def y(self):
        return self.__y

    def __iter__(self):
        return (i for i in (self.x, self.y))

    def __repr__(self):
       class_name = type(self).__name__
        return '{}({!r}, {!r})'.format(class_name, *self)

    def __str__(self):
        return str(tuple(self))

    def __bytes__(self):
        return (bytes([ord(self.typecode)]) +
                bytes(array(self.typecode, self)))

    def __eq__(self, other):
        return tuple(self) == tuple(other)

    def __hash__(self):
        return hash(self.x) ^ hash(self.y)

    def __abs__(self):
        return math.hypot(self.x, self.y)

    def __bool__(self):
        return bool(abs(self))

    def angle(self):
        return math.atan2(self.y, self.x)

    def __format__(self, fmt_spec=''):
        if fmt_spec.endswith('p'):
            fmt_spec = fmt_spec[:-1]
        coords = (abs(self), self.angle())
        outer_fmt = '<{}, {}>'
    else:
        coords = self
        outer_fmt = '({}, {})'
```

```
        components = (format(c, fmt_spec) for c in coords)
        return outer_fmt.format(*components)

    @classmethod
    def frombytes(cls, octets):
        typecode = chr(octets[0])
        memv = memoryview(octets[1:]).cast(typecode)
        return cls(*memv)
```

小结一下，前两节说明了一些特殊方法，要想得到功能完善的对象，这些方法可能是必备的。当然，如果你的应用用不到，就没必要全部实现这些方法。客户并不关心你的对象是否符合 Python 风格。

示例 9-9 中定义的 `Vector2d` 类只是为了教学，我们为它定义了许多与对象表示形式有关的特殊方法。不是每个用户自定义的类都要这样做。

下一节暂时不继续定义 `Vector2d` 类了，我们将讨论 Python 对私有属性（带两个下划线前缀的属性，如 `self.__x`）的设计方式及其缺点。

9.7　Python的私有属性和“受保护的”属性

Python 不能像 Java 那样使用 `private` 修饰符创建私有属性，但是 Python 有个简单的机制，能避免子类意外覆盖“私有”属性。

举个例子。有人编写了一个名为 `Dog` 的类，这个类的内部用到了 `mood` 实例属性，但是没有将其开放。现在，你创建了 `Dog` 类的子类：`Beagle`。如果你在毫不知情的情况下又创建了名为 `mood` 的实例属性，那么在继承的方法中就会把 `Dog` 类的 `mood` 属性覆盖掉。这是个难以调试的问题。

为了避免这种情况，如果以 `__mood` 的形式（两个前导下划线，尾部没有或最多有一个下划线）命名实例属性，Python 会把属性名存入实例的 `__dict__` 属性中，而且会在前面加上一个下划线和类名。因此，对 `Dog` 类来说，`__mood` 会变成 `_Dog__mood`；对 `Beagle` 类来说，会变成 `_Beagle__mood`。这个语言特性叫名称改写（name mangling）。

示例 9-10 以示例 9-7 中定义的 `Vector2d` 类为例来说明名称改写。

示例 **9-10**　私有属性的名称会被“改写”，在前面加上下划线和类名

```
>>> v1 = Vector2d(3, 4)
>>> v1.__dict__
{'_Vector2d__y': 4.0, '_Vector2d__x': 3.0}
>>> v1._Vector2d__x
3.0
```

名称改写是一种安全措施，不能保证万无一失：它的目的是避免意外访问，不能防止故意做错事（图 9-1 也是一种保护装置）。

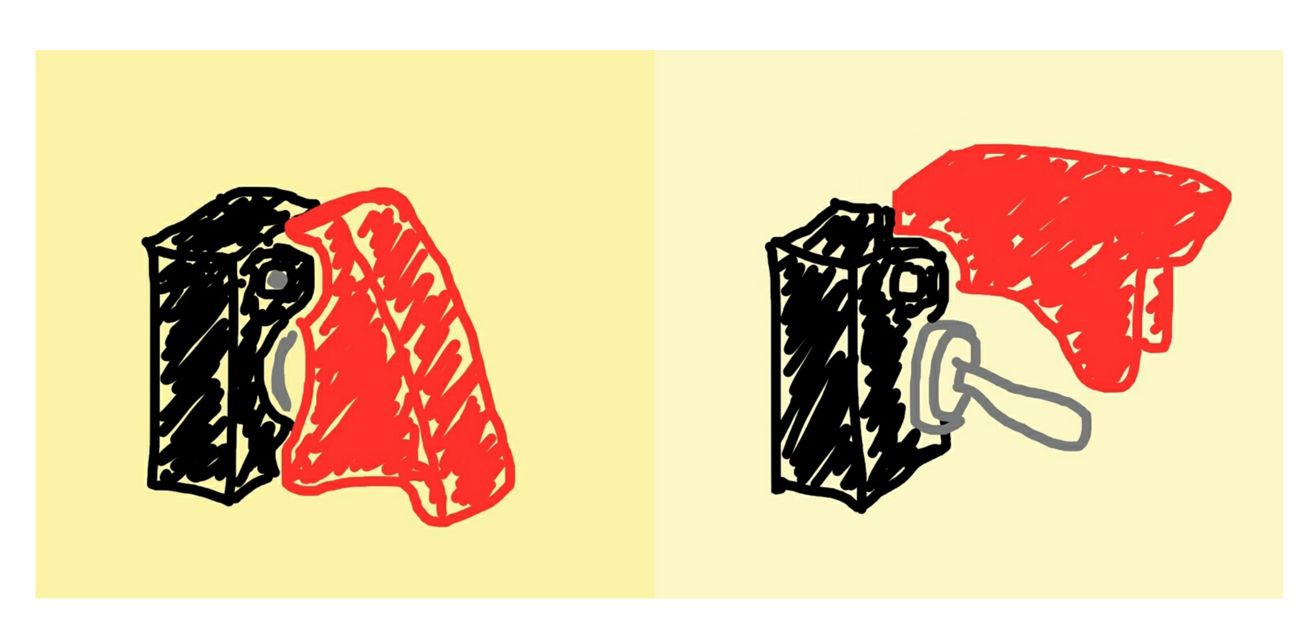


图 **9-1**：把手上的盖子是种保护装置，而不是安全装置：它能避免意外触动把手，但是不能防止有意转动

如示例 9-10 中的最后一行所示，只要知道改写私有属性名的机制，任何人都能直接读取私有属性——这对调试和序列化倒是有用。此外，只要编写 `v1._Vector__x = 7` 这样的代码，就能轻松地为 `Vector2d` 实例的私有分量直接赋值。如果真在生产环境中这么做了，出问题时可别抱怨。

不是所有 Python 程序员都喜欢名称改写功能，也不是所有人都喜欢 `self.__x` 这种不对称的名称。有些人不喜欢这种句法，他们约定使用一个下划线前缀编写“受保护”的属性（如 `self._x`）。批评使用两个下划线这种改写机制的人认为，应该使用命名约定来避免意外覆盖属性。本章开头引用了多产的 Ian Bicking 的一句话，那句话的完整表述如下：

> 绝对不要使用两个前导下划线，这是很烦人的自私行为。如果担心名称冲突，应该明确使用一种名称改写方式（如 `_MyThing_blahblah`）。这其实与使用双下划线一样，不过自己定的规则比双下划线易于理解。[7]

[7]摘自 Paste 的风格指南（http://pythonpaste.org/StyleGuide.html）。

Python 解释器不会对使用单个下划线的属性名做特殊处理，不过这是很多 Python 程序员严格遵守的约定，他们不会在类外部访问这种属性。[8]

遵守使用一个下划线标记对象的私有属性很容易，就像遵守使用全大写字母编写常量那样容易。

> [8]不过在模块中，顶层名称使用一个前导下划线的话，的确会有影响：对 `from mymod import *` 来说，`mymod` 中前缀为下划线的名称不会被导入。然而，依旧可以使用 `from mymod import _privatefunc` 将其导入。Python 教程的 6.1 节“More on Modules”（https://docs.python.org/3/tutorial/modules.html#more-on-modules）说明了这一点。

Python 文档的某些角落把使用一个下划线前缀标记的属性称为“受保护的”属性。[9] 使用 `self._x` 这种形式保护属性的做法很常见，但是很少有人把这种属性叫作“受保护的”属性。有些人甚至将其称为“私有”属性。

> [9]`gettext` 模块中就有一个例子（https://docs.python.org/3/library/gettext.html#gettext.NullTranslations）。

总之，`Vector2d` 的分量都是“私有的”，而且 `Vector2d` 实例都是“不可变的”。我用了两对引号，这是因为并不能真正实现私有和不可变。[10]

> [10]如果这个说法让你感到沮丧，而且让你觉得在这方面 Python 应该向 Java 看齐的话，那么别去读本章的“杂谈”，我在其中对 Java 的 `private` 修饰符的相对强度进行了探讨。

下面继续定义 `Vector2d` 类。在最后一节中，我们将讨论一个特殊的属性（不是方法），它会影响对象的内部存储，对内存用量可能也有重大影响，不过对对象的公开接口没什么影响。这个属性是 `__slots__`。

9.8　使用 `__slots__` 类属性节省空间

默认情况下，Python 在各个实例中名为 `__dict__` 的字典里存储实例属性。如 3.9.3 节所述，为了使用底层的散列表提升访问速度，字典会消耗大量内存。如果要处理数百万个属性不多的实例，通过 `__slots__` 类属性，能节省大量内存，方法是让解释器在元组中存储实例属性，而不用字典。

> 继承自超类的 `__slots__` 属性没有效果。Python 只会使用各个类中定义的 `__slots__` 属性。

定义 `__slots__` 的方式是，创建一个类属性，使用 `__slots__` 这个名字，并把它的值设为一个字符串构成的可迭代对象，其中各个元素表示各个实例属性。我喜欢使用元组，因为这样定义的 `__slots__` 中所含的信息不会变化，如示例 9-11 所示。

示例 **9-11**　vector2d_v3_slots.py：只在 `Vector2d` 类中添加了 `__slots__` 属性

```
class Vector2d:
    __slots__ = ('__x', '__y')

    typecode = 'd'

    # 下面是各个方法（因排版需要而省略了）
```

在类中定义 `__slots__` 属性的目的是告诉解释器："这个类中的所有实例属性都在这儿了！"这样，Python 会在各个实例中使用类似元组的结构存储实例变量，从而避免使用消耗内存的 `__dict__` 属性。如果有数百万个实例同时活动，这样做能节省大量内存。

> 如果要处理数百万个数值对象，应该使用 NumPy 数组（参见 2.9.3 节）。NumPy 数组能高效使用内存，而且提供了高度优化的

数值处理函数，其中很多都一次操作整个数组。我定义 `Vector2d` 类的目的是讨论特殊方法，因为我不太想随便举些例子。

在示例 9-12 中，我们运行了两个构建列表的脚本，这两个脚本都使用列表推导创建 10 000 000 个 `Vector2d` 实例。mem_test.py 脚本的命令行参数是一个模块的名字，模块中定义了不同版本的 `Vector2d` 类。第一次运行使用的是 `vector2d_v3.Vector2d` 类（在示例 9-7 中定义），第二次运行使用的是定义了 `__slots__` 的 `vector2d_v3_slots.Vector2d` 类。

示例 **9-12**　mem_test.py 使用指定模块（如 vector2d_v3.py）中定义的 `Vector2d` 类创建 10 000 000 个实例

```
$ time python3 mem_test.py vector2d_v3.py
Selected Vector2d type: vector2d_v3.Vector2d
Creating 10,000,000 Vector2d instances
Initial RAM usage:       5,623,808
  Final RAM usage:   1,558,482,944

real  0m16.721s
user  0m15.568s
sys 0m1.149s
$ time python3 mem_test.py vector2d_v3_slots.py
Selected Vector2d type: vector2d_v3_slots.Vector2d
Creating 10,000,000 Vector2d instances
Initial RAM usage:      5,718,016
  Final RAM usage:    655,466,496

real   0m13.605s
user   0m13.163s
sys 0m0.434s
```

如示例 9-12 所示，在 10 000 000 个 `Vector2d` 实例中使用 `__dict__` 属性时，RAM 用量高达 1.5GB；而在 `Vector2d` 类中定义 `__slots__` 属性之后，RAM 用量降到了 655MB。此外，定义了 `__slots__` 属性的版本运行速度也更快。这个测试中使用的 mem_test.py 脚本其实只用于加载一个模块、检查内存用量和格式化结果，所用的代码与本章没有太大关联，因此放入附录 A 中的示例 A-4 里。

在类中定义 `__slots__` 属性之后，实例不能再有 `__slots__` 中所列名称之外的其他属性。这只是一个副作用，不是 `__slots__` 存在的真正原因。不要使用 `__slots__` 属性禁止类的用户新增实例属性。`__slots__` 是用于优化的，不是为了约束程序员。

然而，“节省的内存也可能被再次吃掉”：如果把 `'__dict__'` 这个名称添加到 `__slots__` 中，实例会在元组中保存各个实例的属性，此外还支持动态创建属性，这些属性存储在常规的 `__dict__` 中。当然，把 `'__dict__'` 添加到 `__slots__` 中可能完全违背了初衷，这取决于各个实例的静态属性和动态属性的数量及其用法。粗心的优化甚至比提早优化还糟糕。

此外，还有一个实例属性可能需要注意，即 `__weakref__` 属性，为了让对象支持弱引用（参见 8.6 节），必须有这个属性。用户定义的类中默认就有 `__weakref__` 属性。可是，如果类中定义了 `__slots__` 属性，而且想把实例作为弱引用的目标，那么要把 `'__weakref__'` 添加到 `__slots__` 中。

综上，`__slots__` 属性有些需要注意的地方，而且不能滥用，不能使用它限制用户能赋值的属性。处理列表数据时 `__slots__` 属性最有用，例如模式固定的数据库记录，以及特大型数据集。然而，如果你经常处理大量数据，一定要了解一下 NumPy（http://www.numpy.org）；此外，数据分析库 pandas（http://pandas.pydata.org）也值得了解，这个库可以处理非数值数据，而且能导入 / 导出很多不同的列表数据格式。

`__slots__` 的问题

总之，如果使用得当，`__slots__` 能显著节省内存，不过有几点要注意。

- 每个子类都要定义 `__slots__` 属性，因为解释器会忽略继承的 `__slots__` 属性。
- 实例只能拥有 `__slots__` 中列出的属性，除非把 `'__dict__'` 加入 `__slots__` 中（这样做就失去了节省内存的功效）。

- 如果不把 `'__weakref__'` 加入 `__slots__`，实例就不能作为弱引用的目标。

如果你的程序不用处理数百万个实例，或许不值得费劲去创建不寻常的类，那就禁止它创建动态属性或者不支持弱引用。与其他优化措施一样，仅当权衡当下的需求并仔细搜集资料后证明确实有必要时，才应该使用 `__slots__` 属性。

本章最后一个话题讨论如何在实例和子类中覆盖类属性。

9.9　覆盖类属性

Python 有个很独特的特性：类属性可用于为实例属性提供默认值。Vector2d 中有个 typecode 类属性，__bytes__ 方法两次用到了它，而且都故意使用 self.typecode 读取它的值。因为 Vector2d 实例本身没有 typecode 属性，所以 self.typecode 默认获取的是 Vector2d.typecode 类属性的值。

但是，如果为不存在的实例属性赋值，会新建实例属性。假如我们为 typecode 实例属性赋值，那么同名类属性不受影响。然而，自此之后，实例读取的 self.typecode 是实例属性 typecode，也就是把同名类属性遮盖了。借助这一特性，可以为各个实例的 typecode 属性定制不同的值。

Vector2d.typecode 属性的默认值是 'd'，即转换成字节序列时使用 8 字节双精度浮点数表示向量的各个分量。如果在转换之前把 Vector2d 实例的 typecode 属性设为 'f'，那么使用 4 字节单精度浮点数表示各个分量，如示例 9-13 所示。

> 我们在讨论如何添加自定义的实例属性，因此示例 9-13 使用的是示例 9-9 中不带 __slots__ 属性的 Vector2d 类。

示例 **9-13**　设定从类中继承的 typecode 属性，自定义一个实例属性

```
>>> from vector2d_v3 import Vector2d
>>> v1 = Vector2d(1.1, 2.2)
>>> dumpd = bytes(v1)
>>> dumpd
b'd\x9a\x99\x99\x99\x99\x99\xf1?\x9a\x99\x99\x99\x99\x99\x01@'
>>> len(dumpd)  # ❶
17
>>> v1.typecode = 'f'  # ❷
>>> dumpf = bytes(v1)
```

```
>>> dumpf
b'f\xcd\xcc\x8c?\xcd\xcc\x0c@'
>>> len(dumpf)  # ❸
9
>>> Vector2d.typecode  # ❹
'd'
```

❶ 默认的字节序列长度为 17 个字节。

❷ 把 `v1` 实例的 `typecode` 属性设为 `'f'`。

❸ 现在得到的字节序列是 9 个字节长。

❹ `Vector2d.typecode` 属性的值不变，只有 `v1` 实例的 `typecode` 属性使用 `'f'`。

现在你应该知道为什么要在得到的字节序列前面加上 `typecode` 的值了：为了支持不同的格式。

如果想修改类属性的值，必须直接在类上修改，不能通过实例修改。如果想修改所有实例（没有 `typecode` 实例变量）的 `typecode` 属性的默认值，可以这么做：

```
>>> Vector2d.typecode = 'f'
```

然而，有种修改方法更符合 Python 风格，而且效果持久，也更有针对性。类属性是公开的，因此会被子类继承，于是经常会创建一个子类，只用于定制类的数据属性。Django 基于类的视图就大量使用了这个技术。具体做法如示例 9-14 所示。

示例 **9-14**　`ShortVector2d` 是 `Vector2d` 的子类，只用于覆盖 `typecode` 的默认值

```
>>> from vector2d_v3 import Vector2d
>>> class ShortVector2d(Vector2d):  # ❶
...     typecode = 'f'
...
>>> sv = ShortVector2d(1/11, 1/27)  # ❷
>>> sv
ShortVector2d(0.09090909090909091, 0.037037037037037035)  # ❸
```

```
>>> len(bytes(sv))  # ❹
9
```

❶ 把 ShortVector2d 定义为 Vector2d 的子类，只用于覆盖 typecode 类属性。

❷ 为了演示，创建一个 ShortVector2d 实例，即 sv。

❸ 查看 sv 的 repr 表示形式。

❹ 确认得到的字节序列长度为 9 字节，而不是之前的 17 字节。

这也说明了我在 Vecto2d.__repr__ 方法中为什么没有硬编码 class_name 的值，而是使用 type(self).__name__ 获取，如下所示：

```
    # 在Vector2d类中定义

    def __repr__(self):
        class_name = type(self).__name__
        return '{}({!r}, {!r})'.format(class_name, *self)
```

如果硬编码 class_name 的值，那么 Vector2d 的子类（如 ShortVector2d）要覆盖 __repr__ 方法，只是为了修改 class_name 的值。从实例的类型中读取类名，__repr__ 方法就可以放心继承。

至此，我们通过一个简单的类说明了如何利用数据模型处理 Python 的其他功能：提供不同的对象表示形式、实现自定义的格式代码、公开只读属性，以及通过 hash() 函数支持集合和映射。

9.10　本章小结

本章的目的是说明，如何使用特殊方法和约定的结构，定义行为良好且符合 Python 风格的类。

vector2d_v3.py（示例 9-9）比 vector2d_v0.py（示例 9-2）更符合 Python 风格吗？ vector2d_v3.py 中的 `Vector2d` 类用到的 Python 功能肯定要多，但是 `Vector2d` 类的第一版和最后一版相比哪个更符合风格，要看使用的上下文。Tim Peter 写的“Python 之禅”说道：

> 简洁胜于复杂。

符合 Python 风格的对象应该正好符合所需，而不是堆砌语言特性。

我不断改写 `Vector2d` 类是为了提供上下文，以便讨论 Python 的特殊方法和编程约定。回看表 1-1，你会发现本章的几个代码清单说明了下述特殊方法。

- 所有用于获取字符串和字节序列表示形式的方法：`__repr__`、`__str__`、`__format__` 和 `__bytes__`。
- 把对象转换成数字的几个方法：`__abs__`、`__bool__`和 `__hash__`。
- 用于测试字节序列转换和支持散列（连同 `__hash__` 方法）的 `__eq__` 运算符。

为了转换成字节序列，我们还实现了一个备选构造方法，即 `Vector2d.frombytes()`，顺便又讨论了 `@classmethod`（十分有用）和 `@staticmethod`（不太有用，使用模块层函数更简单）两个装饰器。`frombytes` 方法的实现方式借鉴了 `array.array` 类中的同名方法。

我们了解到，格式规范微语言（https://docs.python.org/3/library/string.html#formatspec）是可扩展的，方法是实现 `__format__` 方法，对提供给内置函数 `format(obj,`

format_spec) 的 format_spec，或者提供给 str.format 方法的 '{:«format_spec»}' 位于代换字段中的 «format_spec» 做简单的解析。

为了把 Vector2d 实例变成可散列的，我们先让它们不可变，至少要把 x 和 y 设为私有属性，再以只读特性公开，以防意外修改它们。随后，我们实现了 __hash__ 方法，使用推荐的异或运算符计算实例属性的散列值。

接着，我们讨论了如何使用 __slots__ 属性节省内存，以及这么做要注意的问题。__slots__ 属性有点棘手，因此仅当处理特别多的实例（数百万个，而不是几千个）时才建议使用。

最后，我们说明了如何通过访问实例属性（如 self.typecode）覆盖类属性。我们先创建一个实例属性，然后创建子类，在类中覆盖类属性。

本章多次提到，我编写代码的方式是为了举例说明如何编写标准 Python 对象的 API。如果用一句话总结本章的内容，那就是：

> 要构建符合 Python 风格的对象，就要观察真正的 Python 对象的行为。
>
> ——古老的中国谚语

9.11　延伸阅读

本章介绍了数据模型的几个特殊方法，因此主要参考资料与第 1 章一样，阅读那些资料能对这个话题有个整体了解。方便起见，我再次给出之前推荐的四个资料，同时再多加几个。

Python 语言参考手册中的"Data Model"一章（https://docs.python.org/3/reference/datamodel.html）

本章用到的方法大部分见于"3.3.1. Basic customization"（https://docs.python.org/3/reference/datamodel.html#basic-customization）。

《Python 技术手册（第 2 版）》，Alex Martelli 著

虽然这本书只涵盖 Python 2.5（第 2 版），但是对数据模型做了深入说明。基本的概念都是一样的，而且自 Python 2.2 起（这一版的内置类型和用户定义的类兼容性变得更好），数据模型的大多数 API 完全没变。

《Python Cookbook（第 3 版）中文版》，David Beazley 和 Brian K. Jones 著

通过诀窍来演示现代化的编程实践。尤其是第 8 章"类与对象"，其中有好几个方案与本章讨论的话题有关。

《Python 参考手册（第 4 版）》，David Beazley 著

详细说明了 Python 2.6 和 Python 3 的数据模型。

本章涵盖了与对象表示形式有关的全部特殊方法，唯有 `__index__` 除外。这个方法的作用是强制把对象转换成整数索引，在特定的序列切片场景中使用，以及满足 NumPy 的一个需求。在实际编程中，你我都不用实现 `__index__` 方法，除非决定新建一种数值类型，并想把它作为参数传给 `__getitem__` 方法。如果好奇的话，可以阅读 A.M.Kuchling 写的"What's New in Python 2.5"（https://docs.python.org/2.5/whatsnew/pep-

357.html），这篇文章做了简要说明；此外，还可以阅读“PEP 357—Allowing Any Object to be Used for Slicing”（https://www.python.org/dev/peps/pep-0357/），这份 PEP 从 C 语言扩展的实现者和 NumPy 的作者 Travis Oliphant 的角度详述了对 `__index__` 方法的需求。

意识到应该区分字符串表示形式的早期语言是 Smalltalk。1996 年，Bobby Woolf 写了一篇题为“How to Display an Object as a String: printString and displayString”的文章（http://esug.org/data/HistoricalDocuments/TheSmalltalkReport/ST07/04wo.p 他在这篇文章中讨论了 Smalltalk 对 `printString` 和 `displayString` 方法的实现。在 9.1 节说明 `repr()` 和 `str()` 的作用时，我从这篇文章中借用了言简意赅的表述，即“便于开发者理解的方式”和“便于用户理解的方式”。

杂谈

特性有助于减少前期投入

在 `Vector2d` 类的第一版中，`x` 和 `y` 属性是公开的；默认情况下，Python 的所有实例属性和类属性都是公开的。这对向量来说是合理的，因为我们要能访问分量。虽然这些向量是可迭代的对象，而且可以拆包成一对变量，但是还要能够通过 `my_vector.x` 和 `my_vector.y` 获取各个分量。

如果觉得应该避免意外更新 `x` 和 `y` 属性，可以实现特性，但是代码的其他部分没有变化，`Vector2d` 的公开接口也不受影响，这一点从 doctest 中可以得知。我们依然能够访问 `my_vector.x` 和 `my_vector.y`。

这表明我们可以先以最简单的方式定义类，也就是使用公开属性，因为如果以后需要对读值方法和设值方法增加控制，那就可以实现特性，这样做对一开始通过公开属性的名称（如 `x` 和 `y`）与对象交互的代码没有影响。

Java 语言采用的方式则截然相反：Java 程序员不能先定义简单的公开属性，然后在需要时再实现特性，因为 Java 语言没有特性。因

此，在 Java 中编写读值方法和设值方法是常态，就算这些方法没做什么有用的事情也得这么做，因为 API 不能从简单的公开属性变成读值方法和设值方法，同时又不影响使用那些属性的代码。

此外，本书的技术审校 Alex Martelli 指出，到处都使用读值方法和设值方法是愚蠢的行为。如果想编写下面的代码：

```
---
>>> my_object.set_foo(my_object.get_foo() + 1)
---
```

这样做就行了：

```
---
>>> my_object.foo += 1
---
```

维基的发明人和极限编程先驱 Ward Cunningham 建议问这个问题："做这件事最简单的方法是什么？"意即，我们应该把焦点放在目标上。[11] 提前实现设值方法和读值方法偏离了目标。在 Python 中，我们可以先使用公开属性，然后等需要时再变成特性。

私有属性的安全性和保障性

> Perl 不会强制你保护隐私。你应该待在客厅外，因为你没收到邀请，而不是因为里面有把枪。
>
> ——Larry Wall
> Perl 之父

Python 和 Perl 在很多方面的做法是截然相反的，但是 Larry 和 Guido 似乎都同意要保护对象的隐私。

这些年我教过许多 Java 程序员学习 Python，我发现很多人都对 Java 提供的隐私保障推崇备至。可事实是，Java 的 `private` 和 `protected` 修饰符往往只是为了防止意外（即一种安全措施）。只有使用安全管理器部署应用时才能保障绝对安全，防止恶意访问；

但是，实际上很少有人这么做，即便在企业中也少见。

下面通过一个 Java 类证明这一点（见示例 9-15）。

示例 **9-15**　Confidential.java：一个 Java 类，定义了一个私有字段，名为 `secret`

```
public class Confidential {

    private String secret = "";

    public Confidential(String text) {
        secret = text.toUpperCase();
    }
}
```

在示例 9-15 中，我把 `text` 转换成大写后存入 `secret` 字段。转换成大写是为了表明 `secret` 字段中的值全部是大写的。

我们要使用 Jython 运行 expose.py 脚本才能真正说明问题。那个脚本使用内省（Java 称之为“反射”）获取私有字段的值。expose.py 脚本的代码在示例 9-16 中。

示例 **9-16**　expose.py：一段 Jython 代码，从另一个类中读取一个私有字段

```
import Confidential

message = Confidential('top secret text')
secret_field = Confidential.getDeclaredField('secret')
secret_field.setAccessible(True)  # 攻破防线
print 'message.secret =', secret_field.get(message)
```

运行示例 9-16 得到的结果如下：

```
$ jython expose.py
message.secret = TOP SECRET TEXT
```

字符串 `'TOP SECRET TEXT'` 从 `Confidential` 类的私有字段 `secret` 中读取。

这里没有什么黑魔法：expose.py 脚本使用 Java 反射 API 获取私有字段 `'secret'` 的引用，然后调用 `'secret_field.setAccessible(True)'` 把它设为可读的。显然，使用 Java 代码也能做到这一点（不过所需的代码行数是这里的三倍多，参见本书代码仓库里的 Expose.java 文件，https://github.com/fluentpython/example-code）。

如果这个 Jython 脚本或 Java 主程序（ 如 `Expose.class`） 在 SecurityManager（http://docs.oracle.com/javase/tutorial/essential/environm
的监管下运行，`.setAccessible(True)` 这个关键的调用就会失败。但是现实中，很少有人部署 Java 应用时会使用 SecurityManager，Java applet 除外（还记得这个吗？）。

我的观点是，Java 中的访问控制修饰符基本上也是安全措施，不能保证万无一失——至少实践中是如此。因此，安心享用 Python 提供的强大功能吧，放心去用吧！

[11]参见“Simplest Thing that Could Possibly Work: A Conversation with Ward Cunningham, Part V”（http://www.artima.com/intv/simplest3.html）。

第 10 章　序列的修改、散列和切片

> 不要检查它是不是鸭子、它的叫声像不像鸭子、它的走路姿势像不像鸭子，等等。具体检查什么取决于你想使用语言的哪些行为。（`comp.lang.python`，2000 年 7 月 26 日）
>
> ——Alex Martelli

本章将以第 9 章定义的二维向量 `Vector2d` 类为基础，向前迈出一大步，定义表示多维向量的 `Vector` 类。这个类的行为与 Python 中标准的不可变扁平序列一样。`Vector` 实例中的元素是浮点数，本章结束后 `Vector` 类将支持下述功能：

- 基本的序列协议—— `__len__` 和 `__getitem__`
- 正确表述拥有很多元素的实例
- 适当的切片支持，用于生成新的 `Vector` 实例
- 综合各个元素的值计算散列值
- 自定义的格式语言扩展

此外，我们还将通过 `__getattr__` 方法实现属性的动态存取，以此取代 `Vector2d` 使用的只读特性——不过，序列类型通常不会这么做。

在大量代码之间，我们将穿插讨论一个概念：把协议当作正式接口。我们将说明协议和鸭子类型之间的关系，以及对自定义类型的实际影响。

我们开始吧！

> 三维以上向量的应用
>
> 谁需要 1000 维向量呢？提示：不是 3D 艺术家！不过，信息检索领

域经常使用 *n* 维向量（*n* 是很大的数），查询的文档和文本使用向量表示，一个单词一个维度。这叫向量空间模型（https://en.wikipedia.org/wiki/Vector_space_model）。在这个模型中，一个关键的相关指标是余弦相关性（即查询向量与文档向量夹角的余弦）。夹角越小，余弦值越趋近于 1，文档与查询的相关性就越大。

不过，本章定义的 `Vector` 类是为了教学而举的例子，不会涉及很多数学原理。我们的目的是以序列类型为背景说明 Python 的几个特殊方法。

如果在实际使用中需要做向量运算，应该使用 NumPy 和 SciPy。Radim Rehurek 开发的 PyPI 包 gensim（https://pypi.python.org/pypi/gensim）使用 NumPy 和 SciPy 实现了用于处理自然语言和检索信息的向量空间模型。

10.1　Vector类：用户定义的序列类型

我们将使用组合模式实现 `Vector` 类，而不使用继承。向量的分量存储在浮点数数组中，而且还将实现不可变扁平序列所需的方法。

不过，在实现序列方法之前，我们要确保 `Vector` 类与前一章定义的 `Vector2d` 类兼容，除非有些地方让二者兼容没有什么意义。

10.2　Vector类第1版：与Vector2d类兼容

Vector 类的第 1 版要尽量与前一章定义的 Vector2d 类兼容。

然而我们会故意不让 Vector 的构造方法与 Vector2d 的构造方法兼容。为了编写 Vector(3, 4) 和 Vector(3, 4, 5) 这样的代码，我们可以让 __init__ 方法接受任意个参数（通过 *args）；但是，序列类型的构造方法最好接受可迭代的对象为参数，因为所有内置的序列类型都是这样做的。示例 10-1 展示了 Vector 类的几种实例化方式。

示例 **10-1**　测试 Vector.__init__ 和 Vector.__repr__ 方法

```
>>> Vector([3.1, 4.2])
Vector([3.1, 4.2])
>>> Vector((3, 4, 5))
Vector([3.0, 4.0, 5.0])
>>> Vector(range(10))
Vector([0.0, 1.0, 2.0, 3.0, 4.0, ...])
```

除了新构造方法的签名外，我还确保了传入两个分量（如 Vector([3, 4])）时，Vector2d 类（如 Vector2d(3, 4)）的每个测试都能通过，而且得到相同的结果。



如果 Vector 实例的分量超过 6 个，repr() 生成的字符串就会使用 ... 省略一部分，如示例 10-1 中的最后一行所示。包含大量元素的集合类型一定要这么做，因为字符串表示形式是用于调试的（因此不想让大型对象在控制台或日志中输出几千行内容）。使用 reprlib 模块可以生成长度有限的表示形式，如示例 10-2 所示。

在 Python 2 中，reprlib 模块的名字是 repr。2to3 工具能自动重写 repr 导入的内容。

示例 10-2 是第 1 版 `Vector` 类的实现代码（以示例 9-2 和示例 9-3 中的代码为基础）。

示例 **10-2**　vector_v1.py：从 vector2d_v1.py 衍生而来

```
from array import array
import reprlib
import math


class Vector:
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components)  ❶

    def __iter__(self):
        return iter(self._components)  ❷

    def __repr__(self):
        components = reprlib.repr(self._components)  ❸
        components = components[components.find('['):-1]  ❹
        return 'Vector({})'.format(components)

    def __str__(self):
        return str(tuple(self))

    def __bytes__(self):
        return (bytes([ord(self.typecode)]) +
                bytes(self._components))  ❺

    def __eq__(self, other):
        return tuple(self) == tuple(other)

    def __abs__(self):
        return math.sqrt(sum(x * x for x in self))  ❻

    def __bool__(self):
        return bool(abs(self))

    @classmethod
    def frombytes(cls, octets):
        typecode = chr(octets[0])
        memv = memoryview(octets[1:]).cast(typecode)
        return cls(memv)  ❼
```

❶ self._components 是“受保护的”实例属性，把 Vector 的分量保存在一个数组中。

❷ 为了迭代，我们使用 self._components 构建一个迭代器。[1]

> [1]iter() 函数和 __iter__ 方法在第 14 章讨论。

❸ 使用 reprlib.repr() 函数获取 self._components 的有限长度表示形式（如 array('d', [0.0, 1.0, 2.0, 3.0, 4.0, ...])）。

❹ 把字符串插入 Vector 的构造方法调用之前，去掉前面的 array('d' 和后面的)。

❺ 直接使用 self._components 构建 bytes 对象。

❻ 不能使用 hypot 方法了，因此我们先计算各分量的平方之和，然后再使用 sqrt 方法开平方。

❼ 我们只需在 Vector2d.frombytes 方法的基础上改动最后一行：直接把 memoryview 传给构造方法，不用像前面那样使用 * 拆包。

我使用 reprlib.repr 的方式需要做些说明。这个函数用于生成大型结构或递归结构的安全表示形式，它会限制输出字符串的长度，用 '...' 表示截断的部分。我希望 Vector 实例的表示形式是 Vector([3.0, 4.0, 5.0]) 这样，而不是 Vector(array('d', [3.0, 4.0, 5.0]))，因为 Vector 实例中的数组是实现细节。因为这两种构造方法的调用方式所构建的 Vector 对象是一样的，所以我选择使用更简单的句法，即传入列表参数。

编写 __repr__ 方法时，本可以使用这个表达式生成简化的 components 显示形式：reprlib.repr(list(self._components))。然而，这么做有点浪费，因为要把 self._components 中的每个元素复制到一个列表中，然后使用列表的表示形式。我没有这么做，而是直接把 self._components 传给 reprlib.repr 函数，然后去掉 [] 外面的字符，如示例 10-2 中 __repr__ 方法的第二行所示。

调用 repr() 函数的目的是调试，因此绝对不能抛出异常。如果 __repr__ 方法的实现有问题，那么必须处理，尽量输出有用的内容，让用户能够识别目标对象。

注意，__str__、__eq__ 和 __bool__ 方法与 Vector2d 类中的一样，而 frombytes 方法也只变了一个字符（最后一行把 * 去掉了）。这是 Vector2d 可迭代的好处之一。

顺便说一下，我们本可以让 Vector 继承 Vector2d，但是我没这么做，原因有二。其一，两个构造方法不兼容，因此不建议继承。这一点可以通过适当处理 __init__ 方法的参数解决，不过第二个原因更重要：我想把 Vector 类当作单独的示例，以此实现序列协议。接下来，我们先讨论协议这个术语，然后实现序列协议。

10.3　协议和鸭子类型

在第 1 章我们就说过，在 Python 中创建功能完善的序列类型无需使用继承，只需实现符合序列协议的方法。不过，这里说的协议是什么呢？

在面向对象编程中，协议是非正式的接口，只在文档中定义，在代码中不定义。例如，Python 的序列协议只需要 `__len__` 和 `__getitem__` 两个方法。任何类（如 `Spam`），只要使用标准的签名和语义实现了这两个方法，就能用在任何期待序列的地方。`Spam` 是不是哪个类的子类无关紧要，只要提供了所需的方法即可。示例 1-1 中见过一例，这里再次给出代码，

如示例 10-3 所示。

示例 **10-3**　示例 1-1 的代码，为了方便，再次给出

```
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                                        for rank in self.ranks]

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, position):
        return self._cards[position]
```

示例 10-3 中的 `FrenchDeck` 类能充分利用 Python 的很多功能，因为它实现了序列协议，不过代码中并没有声明这一点。任何有经验的 Python 程序员只要看一眼就知道它是序列，即便它是 `object` 的子类也无妨。我们说它是序列，因为它的行为像序列，这才是重点。

根据本章开头引用的 Alex Martelli 的帖子，人们称其为鸭子类型（duck typing）。

协议是非正式的，没有强制力，因此如果你知道类的具体使用场景，通常只需要实现一个协议的部分。例如，为了支持迭代，只需实现 `__getitem__` 方法，没必要提供 `__len__` 方法。

下面，我们将在 `Vector` 类中实现序列协议。我们先不支持完美的切片，稍后再完善。

10.4　Vector类第2版：可切片的序列

如 FrenchDeck 类所示，如果能委托给对象中的序列属性（如 self._components 数组），支持序列协议特别简单。下述只有一行代码的 __len__ 和 __getitem__ 方法是个好的开始：

```
class Vector:
    # 省略了很多行
    # ...

    def __len__(self):
        return len(self._components)

    def __getitem__(self, index):
        return self._components[index]
```

添加这两个方法之后，就能执行下述操作了：

```
>>> v1 = Vector([3, 4, 5])
>>> len(v1)
3
>>> v1[0], v1[-1]
(3.0, 5.0)
>>> v7 = Vector(range(7))
>>> v7[1:4]
array('d', [1.0, 2.0, 3.0])
```

可以看到，现在连切片都支持了，不过尚不完美。如果 Vector 实例的切片也是 Vector 实例，而不是数组，那就更好了。前面那个 FrenchDeck 类也有类似的问题：切片得到的是列表。对 Vector 来说，如果切片生成普通的数组，将会缺失大量功能。

想想内置的序列类型，切片得到的都是各自类型的新实例，而不是其他类型。

为了把 Vector 实例的切片也变成 Vector 实例，我们不能简单地委托给数组切片。我们要分析传给 __getitem__ 方法的参数，做适当的处

理。

下面来看 Python 如何把 `my_seq[1:3]` 句法变成传给 `my_seq.__getitem__(...)` 的参数。

10.4.1　切片原理

一例胜千言，我们来看看示例 10-4。

示例 **10-4**　了解 `__getitem__` 和切片的行为

```
>>> class MySeq:
...     def __getitem__(self, index):
...         return index  # ❶
...
>>> s = MySeq()
>>> s[1]  # ❷
1
>>> s[1:4]  # ❸
slice(1, 4, None)
>>> s[1:4:2]  # ❹
slice(1, 4, 2)
>>> s[1:4:2, 9]  # ❺
(slice(1, 4, 2), 9)
>>> s[1:4:2, 7:9]  # ❻
(slice(1, 4, 2), slice(7, 9, None))
```

❶ 在这个示例中，`__getitem__` 直接返回传给它的值。

❷ 单个索引，没什么新奇的。

❸ `1:4` 表示法变成了 `slice(1, 4, None)`。

❹ `slice(1, 4, 2)` 的意思是从 1 开始，到 4 结束，步幅为 2。

❺ 神奇的事发生了：如果 `[]` 中有逗号，那么 `__getitem__` 收到的是元组。

❻ 元组中甚至可以有多个切片对象。

现在，我们来仔细看看 `slice` 本身，如示例 10-5 所示。

示例 **10-5**　查看 `slice` 类的属性

```
>>> slice  # ❶
<class 'slice'>
>>> dir(slice)  # ❷
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__gt__',
 '__hash__', '__init__', '__le__', '__lt__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 'indices', 'start', 'step', 'stop']
```

❶ `slice` 是内置的类型（2.4.2 节首次出现）。

❷ 通过审查 `slice`，发现它有 `start`、`stop` 和 `step` 数据属性，以及 `indices` 方法。

在示例 10-5 中，调用 `dir(slice)` 得到的结果中有个 `indices` 属性，这个方法有很大的作用，但是鲜为人知。`help(slice.indices)` 给出的信息如下。

`S.indices(len) -> (start, stop, stride)`

　　给定长度为 `len` 的序列，计算 `S` 表示的扩展切片的起始（`start`）和结尾（`stop`）索引，以及步幅（`stride`）。超出边界的索引会被截掉，这与常规切片的处理方式一样。

换句话说，`indices` 方法开放了内置序列实现的棘手逻辑，用于优雅地处理缺失索引和负数索引，以及长度超过目标序列的切片。这个方法会“整顿”元组，把 `start`、`stop` 和 `stride` 都变成非负数，而且都落在指定长度序列的边界内。

下面举几个例子。假设有个长度为 5 的序列，例如 `'ABCDE'`：

```
>>> slice(None, 10, 2).indices(5)  # ❶
(0, 5, 2)
>>> slice(-3, None, None).indices(5)  # ❷
(2, 5, 1)
```

❶ 'ABCDE'[:10:2] 等同于 'ABCDE'[0:5:2]

❷ 'ABCDE'[-3:] 等同于 'ABCDE'[2:5:1]

写作本书时，在线版 Python 库参考好像还没有 slice.indices 方法的文档。[2]Python Python/C API 参考手册中有类似的 C 语言函数的文档，PySlice_GetIndicesEx（https://docs.python.org/3/c-api/slice.html#c.PySlice_GetIndicesEx）。研究切片对象时，我在 Python 控制台中执行了 dir() 和 help()，这才发现 slice.indices() 方法。这也表明交互式控制台是个有价值的工具，能发现新事物。

[2]现在已经有了，参见：https://docs.python.org/3/reference/datamodel.html?highlight=indices#slice.indices。——编者注

在 Vector 类中无需使用 slice.indices() 方法，因为收到切片参数时，我们会委托 _components 数组处理。但是，如果你没有底层序列类型作为依靠，那么使用这个方法能节省大量时间。

现在我们知道如何处理切片了，下面来看 Vector.__getitem__ 方法改进后的实现。

10.4.2　能处理切片的__getitem__方法

示例 10-6 列出了让 Vector 表现为序列所需的两个方法：__len__ 和 __getitem__ （后者现在能正确地处理切片了）。

示例 **10-6**　vector_v2.py 的部分代码：为 vector_v1.py 中的 Vector 类（见示例 10-2）添加 __len__ 和__getitem__ 方法

```
    def __len__(self):
        return len(self._components)

    def __getitem__(self, index):
        cls = type(self)  ❶
        if isinstance(index, slice):  ❷
```

```
            return cls(self._components[index])  ❸
        elif isinstance(index, numbers.Integral):  ❹
            return self._components[index]  ❺
        else:
            msg = '{cls.__name__} indices must be integers'
            raise TypeError(msg.format(cls=cls))  ❻
```

❶ 获取实例所属的类（即 `Vector`），供后面使用。

❷ 如果 `index` 参数的值是 `slice` 对象……

❸ ……调用类的构造方法，使用 `_components` 数组的切片构建一个新 `Vector` 实例。

❹ 如果 `index` 是 `int` 或其他整数类型……[3]

> [3]必须在 vector_v2.py 的开头加上 `import numbers`。——编者注

❺ ……那就返回 `_components` 中相应的元素。

❻ 否则，抛出异常。

> 大量使用 `isinstance` 可能表明面向对象设计得不好，不过在 `__getitem__` 方法中使用它处理切片是合理的。注意，示例 10-6 中测试时用的是 `numbers.Integral`，这是一个抽象基类（Abstract Base Class，ABC）。在 `isinstance` 中使用抽象基类做测试能让 API 更灵活且更容易更新，原因参见第 11 章。可惜，Python 3.4 的标准库中没有 `slice` 的抽象基类。

为了确定在 `__getitem__` 的 `else` 子句中会抛出哪个异常，我在交互式控制台中查看了 `'ABC'[1, 2]` 的结果。我发现，Python 抛出的是 `TypeError`；我还从错误消息中复制了表述方式，“indices must be integers”。为了创建符合 Python 风格的对象，我们要模仿 Python 内置的对象。

把示例 10-6 中的代码添加到 `Vector` 类中之后，切片行为就正确了，如示例 10-7 所示。

示例 **10-7**　测试示例 10-6 中改进的 `Vector.__getitem__` 方法

```
>>> v7 = Vector(range(7))
>>> v7[-1] ❶
6.0
>>> v7[1:4] ❷
Vector([1.0, 2.0, 3.0])
>>> v7[-1:] ❸
Vector([6.0])
>>> v7[1,2] ❹
Traceback (most recent call last):
  ...
TypeError: Vector indices must be integers
```

❶ 单个整数索引只获取一个分量，值为浮点数。

❷ 切片索引创建一个新 `Vector` 实例。

❸ 长度为 `1` 的切片也创建一个 `Vector` 实例。

❹ `Vector` 不支持多维索引，因此索引元组或多个切片会抛出错误。

10.5　Vector类第3版：动态存取属性

Vector2d 变成 Vector 之后，就没办法通过名称访问向量的分量了（如 v.x 和 v.y）。现在我们处理的向量可能有大量分量。不过，若能通过单个字母访问前几个分量的话会比较方便。比如，用 x、y 和 z 代替 v[0]、v[1] 和 v[2]。

我们想额外提供下述句法，用于读取向量的前四个分量：

```
>>> v = Vector(range(10))
>>> v.x
0.0
>>> v.y, v.z, v.t
(1.0, 2.0, 3.0)
```

在 Vector2d 中，我们使用 @property 装饰器把 x 和 y 标记为只读特性（见示例 9-7）。我们可以在 Vector 中编写四个特性，但这样太麻烦。特殊方法 __getattr__ 提供了更好的方式。

属性查找失败后，解释器会调用 __getattr__ 方法。简单来说，对 my_obj.x 表达式，Python 会检查 my_obj 实例有没有名为 x 的属性；如果没有，到类（my_obj.__class__）中查找；如果还没有，顺着继承树继续查找。[4] 如果依旧找不到，调用 my_obj 所属类中定义的 __getattr__ 方法，传入 self 和属性名称的字符串形式（如 'x'）。

> [4]属性查找机制比这复杂得多，复杂的细节在第六部分讲解。目前知道这种简单的说明即可。

示例 10-8 中列出的是我们为 Vector 类定义的 __getattr__ 方法。这个方法的作用很简单，它检查所查找的属性是不是 xyzt 中的某个字母，如果是，那么返回对应的分量。

示例 **10-8**　vector_v3.py 的部分代码：在 vector_v2.py 中定义的 Vector 类里添加 __getattr__ 方法

```
shortcut_names = 'xyzt'
```

```
def __getattr__(self, name):
    cls = type(self)  ❶

    if len(name) == 1:  ❷
        pos = cls.shortcut_names.find(name)  ❸
        if 0 <= pos < len(self._components):  ❹
            return self._components[pos]
    msg = '{.__name__!r} object has no attribute {!r}'  ❺
    raise AttributeError(msg.format(cls, name))
```

❶ 获取 `Vector`，后面待用。

❷ 如果属性名只有一个字母，可能是 `shortcut_names` 中的一个。

❸ 查找那个字母的位置；`str.find` 还会定位 `'yz'`，但是我们不需要，因此在前一行做了测试。

❹ 如果位置落在范围内，返回数组中对应的元素。

❺ 如果测试都失败了，抛出 `AttributeError`，并指明标准的消息文本。

`__getattr__` 方法的实现不难，但是这样实现还不够。看看示例 `10-9` 中古怪的交互行为。

示例 **10-9**　不恰当的行为：为 `v.x` 赋值没有抛出错误，但是前后矛盾

```
>>> v = Vector(range(5))
>>> v
Vector([0.0, 1.0, 2.0, 3.0, 4.0])
>>> v.x  # ❶
0.0
>>> v.x = 10  # ❷
>>> v.x  # ❸
10
>>> v
Vector([0.0, 1.0, 2.0, 3.0, 4.0])  # ❹
```

❶ 使用 `v.x` 获取第一个元素（`v[0]`）。

❷ 为 `v.x` 赋新值。这个操作应该抛出异常。

❸ 读取 `v.x`，得到的是新值，`10`。

❹ 可是，向量的分量没变。

你能解释为什么会这样吗？具体而言，如果向量的分量数组中没有新值，为什么 `v.x` 返回 `10`？如果你不能立即给出解释，再看看示例 10-8 前面对 `__getattr__` 方法的说明。原因不是很明显，但却是理解本书后面内容的重要基础。

示例 10-9 之所以前后矛盾，是 `__getattr__` 的运作方式导致的：仅当对象没有指定名称的属性时，Python 才会调用那个方法，这是一种后备机制。可是，像 `v.x = 10` 这样赋值之后，`v` 对象有 `x` 属性了，因此使用 `v.x` 获取 `x` 属性的值时不会调用 `__getattr__` 方法了，解释器直接返回绑定到 `v.x` 上的值，即 `10`。另一方面，`__getattr__` 方法的实现没有考虑到 `self._components` 之外的实例属性，而是从这个属性中获取 `shortcut_names` 中所列的“虚拟属性”。

为了避免这种前后矛盾的现象，我们要改写 `Vector` 类中设置属性的逻辑。

回想第 9 章的最后一个 `Vector2d` 示例中，如果为 `.x` 或 `.y` 实例属性赋值，会抛出 `AttributeError`。为了避免歧义，在 `Vector` 类中，如果为名称是单个小写字母的属性赋值，我们也想抛出那个异常。为此，我们要实现 `__setattr__` 方法，如示例 10-10 所示。

示例 **10-10**　vector_v3.py 的部分代码：在 `Vector` 类中实现 `__setattr__` 方法

```
def __setattr__(self, name, value):
    cls = type(self)
    if len(name) == 1:  ❶
        if name in cls.shortcut_names:  ❷
            error = 'readonly attribute {attr_name!r}'
        elif name.islower():  ❸
            error = "can't set attributes 'a' to 'z' in {cls_name!r}"
        else:
            error = ''  ❹
```

```
        if error:  ❺
            msg = error.format(cls_name=cls.__name__, attr_name=name)
            raise AttributeError(msg)
    super().__setattr__(name, value)  ❻
```

❶ 特别处理名称是单个字符的属性。

❷ 如果 `name` 是 `xyzt` 中的一个，设置特殊的错误消息。

❸ 如果 `name` 是小写字母，为所有小写字母设置一个错误消息。

❹ 否则，把错误消息设为空字符串。

❺ 如果有错误消息，抛出 `AttributeError`。

❻ 默认情况：在超类上调用 `__setattr__` 方法，提供标准行为。

> `super()` 函数用于动态访问超类的方法，对 Python 这样支持多重继承的动态语言来说，必须能这么做。程序员经常使用这个函数把子类方法的某些任务委托给超类中适当的方法，如示例 10-10 所示。12.2 节会进一步探讨 `super()` 函数。

为了给 `AttributeError` 选择错误消息，我查看了内置的 `complex` 类型的行为，因为 `complex` 对象是不可变的，而且有一对数据属性：`real` 和 `imag`。如果试图修改任何一个属性，`complex` 实例会抛出 `AttributeError`，而且把错误消息设为`"can't set attribute"`。而如果尝试为受特性保护的只读属性赋值（像 9.6 节那样做），得到的错误消息是`"readonly attribute"`。在 `__setitem__` 方法中为 `error` 字符串选词时，我参考了这两个错误消息，而且更为明确地指出了禁止赋值的属性。

注意，我们没有禁止为全部属性赋值，只是禁止为单个小写字母属性赋值，以防与只读属性 `x`、`y`、`z` 和 `t` 混淆。

> 我们知道，在类中声明 `__slots__` 属性可以防止设置新实例属性；因此，你可能想使用这个功能，而不像这里所做的，实现

`__setattr__` 方法。可是，正如 9.8.1 节所指出的，不建议只为了避免创建实例属性而使用 `__slots__` 属性。`__slots__` 属性只应该用于节省内存，而且仅当内存严重不足时才应该这么做。

虽然这个示例不支持为 `Vector` 分量赋值，但是有一个问题要特别注意：多数时候，如果实现了 `__getattr__` 方法，那么也要定义 `__setattr__` 方法，以防对象的行为不一致。

如果想允许修改分量，可以使用 `__setitem__` 方法，支持 `v[0] = 1.1` 这样的赋值，以及（或者）实现 `__setattr__` 方法，支持 `v.x = 1.1` 这样的赋值。不过，我们要保持 `Vector` 是不可变的，因为在下一节中，我们将把它变成可散列的。

10.6　Vector类第4版：散列和快速等值测试

我们要再次实现 `__hash__` 方法。加上现有的 `__eq__` 方法，这会把 `Vector` 实例变成可散列的对象。

示例 9-8 中的 `__hash__` 方法简单地计算 `hash(self.x) ^ hash(self.y)`。这一次，我们要使用^（异或）运算符依次计算各个分量的散列值，像这样：`v[0] ^ v[1] ^ v[2]...`。这正是 `functools.reduce` 函数的作用。之前我说 `reduce` 没有以往那么常用，[5] 但是计算向量所有分量的散列值非常适合使用这个函数。`reduce` 函数的整体思路如图 10-1 所示。

> [5] `sum`、`any` 和 `all` 涵盖了 `reduce` 的大部分用途。参见 5.2.1 节的讨论。

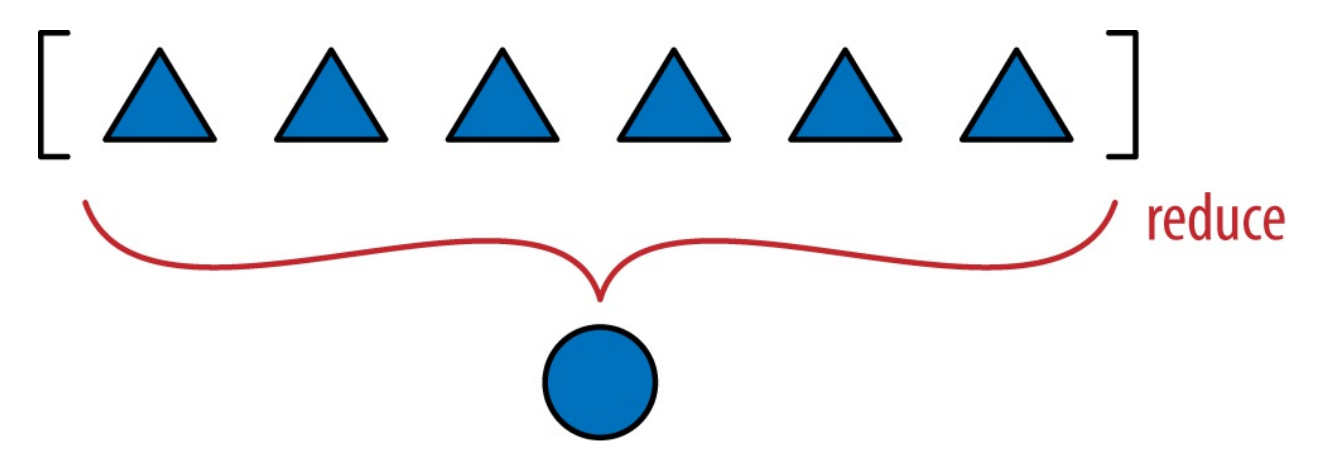


图 **10-1**：归约函数（**`reduce`**、**`sum`**、**`any`**、**`all`**）把序列或有限的可迭代对象变成一个聚合结果

我们已经知道 `functools.reduce()` 可以替换成 `sum()`，下面说说它的原理。它的关键思想是，把一系列值归约成单个值。`reduce()` 函数的第一个参数是接受两个参数的函数，第二个参数是一个可迭代的对象。假如有个接受两个参数的 `fn` 函数和一个 `lst` 列表。调用 `reduce(fn, lst)` 时，`fn` 会应用到第一对元素上，即 `fn(lst[0], lst[1])`，生成第一个结果 `r1`。然后，`fn` 会应用到 `r1` 和下一个元素

上，即 fn(r1, lst[2])，生成第二个结果 r2。接着，调用 fn(r2, lst[3])，生成 r3……直到最后一个元素，返回最后得到的结果 rN。

使用 reduce 函数可以计算 5!（5 的阶乘）：

```
>>> 2 * 3 * 4 * 5  # 想要的结果是：5! == 120
120
>>> import functools
>>> functools.reduce(lambda a,b: a*b, range(1, 6))
120
```

回到散列问题上。示例 10-11 展示了计算聚合异或的 3 种方式：一种使用 for 循环，两种使用 reduce 函数。

示例 **10-11**　计算整数 0~5 的累计异或的 3 种方式

```
>>> n = 0
>>> for i in range(1, 6):  # ❶
...     n ^= i
...
>>> n
1
>>> import functools
>>> functools.reduce(lambda a, b: a^b, range(6))  # ❷
1
>>> import operator
>>> functools.reduce(operator.xor, range(6))  # ❸
1
```

❶ 使用 for 循环和累加器变量计算聚合异或。

❷ 使用 functools.reduce 函数，传入匿名函数。

❸ 使用 functools.reduce 函数，把 lambda 表达式换成 operator.xor。

示例 10-11 中的 3 种方式里，我最喜欢最后一种，其次是 for 循环。你呢？

5.10.1 节讲过，operator 模块以函数的形式提供了 Python 的全部中缀运算符，从而减少使用 lambda 表达式。

为了使用我喜欢的方式编写 `Vector.__hash__` 方法，我们要导入 `functools` 和 `operator` 模块。`Vector` 类的相关变化如示例 10-12 所示。

示例 **10-12**　vector_v4.py 的部分代码：在 vector_v3.py 中 `Vector` 类的基础上导入两个模块，添加 `__hash__` 方法

```
from array import array
import reprlib
import math
import functools  # ❶
import operator  # ❷


class Vector:
    typecode = 'd'

    # 排版需要，省略了很多行...

    def __eq__(self, other):  # ❸
        return tuple(self) == tuple(other)

    def __hash__(self):
        hashes = (hash(x) for x in self._components)  # ❹
        return functools.reduce(operator.xor, hashes, 0)  # ❺

    # 省略了很多行...
```

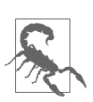
❶ 为了使用 `reduce` 函数，导入 `functools` 模块。

❷ 为了使用 `xor` 函数，导入 `operator` 模块。

❸ `__eq__` 方法没变；我把它列出来是为了把它和 `__hash__` 方法放在一起，因为它们要结合在一起使用。

❹ 创建一个生成器表达式，惰性计算各个分量的散列值。

❺ 把 `hashes` 提供给 `reduce` 函数，使用 `xor` 函数计算聚合的散列值；第三个参数，`0` 是初始值（参见下面的警告框）。



使用 reduce 函数时最好提供第三个参数，reduce(function, iterable, initializer)，这样能避免这个异常：TypeError: reduce() of empty sequence with no initial value（这个错误消息很棒，说明了问题，还提供了解决方法）。如果序列为空，initializer 是返回的结果；否则，在归约中使用它作为第一个参数，因此应该使用恒等值。比如，对 +、| 和 ^ 来说， initializer 应该是 0；而对 * 和 & 来说，应该是 1。

示例 10-12 中实现的 __hash__ 方法是一种映射归约计算（见图 10-2）。

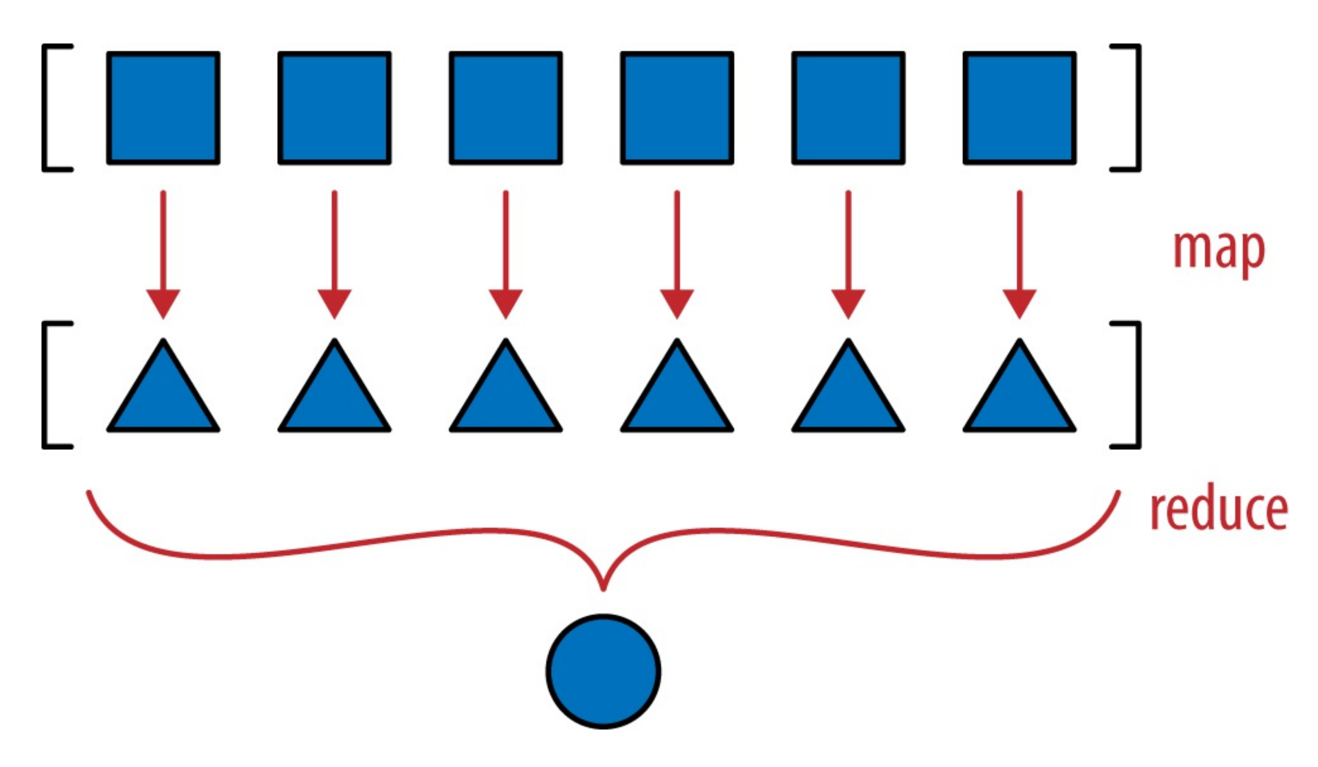


图 **10-2**：映射归约：把函数应用到各个元素上，生成一个新序列（映射，**map**），然后计算聚合值（归约，**reduce**）

映射过程计算各个分量的散列值，归约过程则使用 xor 运算符聚合所有散列值。把生成器表达式替换成 map 方法，映射过程更明显：

```
def __hash__(self):
    hashes = map(hash, self._components)
    return functools.reduce(operator.xor, hashes)
```

在 Python 2 中使用 map 函数效率低些，因为 map 函数要使用结果构建一个列表。但是在 Python 3 中，map 函数是惰性的，它会创建一个生成器，按需产出结果，因此能节省内存——这与示例 10-12 中使用生成器表达式定义 __hash__ 方法的原理一样。

既然讲到了归约函数，那就把前面草草实现的 __eq__ 方法修改一下，减少处理时间和内存用量——至少对大型向量来说是这样。如示例 9-2 所示，__eq__ 方法的实现可以非常简洁：

```
def __eq__(self, other):
    return tuple(self) == tuple(other)
```

Vector2d 和 Vector 都可以这样做，它甚至还会认为 Vector([1, 2]) 和 (1, 2) 相等。这或许是个问题，不过我们暂且忽略。[6] 可是，这样做对有几千个分量的 Vector 实例来说，效率十分低下。上述实现方式要完整复制两个操作数，构建两个元组，而这么做只是为了使用 tuple 类型的 __eq__ 方法。对 Vector2d（只有两个分量）来说，这是个捷径，但是对维数很多的向量来说情况就不同了。示例 10-13 中比较两个 Vector 实例（或者比较一个 Vector 实例与一个可迭代对象）的方式更好。

[6] 13.1 节会认真对待 Vector([1, 2]) == (1, 2) 这个问题。

示例 **10-13** 为了提高比较的效率，Vector.__eq__ 方法在 for 循环中使用 zip 函数

```
def __eq__(self, other):
    if len(self) != len(other):  # ❶
        return False
    for a, b in zip(self, other):  # ❷
        if a != b:  # ❸
            return False
    return True  # ❹
```

❶ 如果两个对象的长度不一样，那么它们不相等。

❷ `zip` 函数生成一个由元组构成的生成器，元组中的元素来自参数传入的各个可迭代对象。如果不熟悉 `zip` 函数，请阅读“出色的 `zip` 函数”附注栏。前面比较长度的测试是有必要的，因为一旦有一个输入耗尽，`zip` 函数会立即停止生成值，而且不发出警告。

❸ 只要有两个分量不同，返回 `False`，退出。

❹ 否则，对象是相等的。

示例 10-13 的效率很好，不过用于计算聚合值的整个 `for` 循环可以替换成一行 `all` 函数调用：如果所有分量对的比较结果都是 `True`，那么结果就是 `True`。只要有一次比较的结果是 `False`，`all` 函数就返回 `False`。使用 `all` 函数实现 `__eq__` 方法的方式如示例 10-14 所示。

示例 **10-14**　使用 `zip` 和 `all` 函数实现 `Vector.__eq__` 方法，逻辑与示例 10-13 一样

```
def __eq__(self, other):
    return len(self) == len(other) and all(a == b for a, b in zip(self, other
```

注意，首先要检查两个操作数的长度是否相同，因为 `zip` 函数会在最短的那个操作数耗尽时停止。

我们选择在 vector_v4.py 中采用示例 10-14 中实现的 `__eq__` 方法。

本章最后要像 `Vector2d` 类那样，为 `Vector` 类实现 `__format__` 方法。

出色的 **`zip`** 函数

使用 `for` 循环迭代元素不用处理索引变量，还能避免很多缺陷，但是需要一些特殊的实用函数协助。其中一个是内置的 `zip` 函数。使用 `zip` 函数能轻松地并行迭代两个或更多可迭代对象，它返回的元组可以拆包成变量，分别对应各个并行输入中的一个元素。如示例 10-15 所示。

zip 函数的名字取自拉链系结物（zipper fastener），因为这个物品用于把两个拉链边的链牙咬合在一起，这形象地说明了 zip(left, right) 的作用。zip 函数与文件压缩没有关系。

示例 **10-15** zip 内置函数的使用示例

```
>>> zip(range(3), 'ABC')  # ❶
<zip object at 0x10063ae48>
>>> list(zip(range(3), 'ABC'))  # ❷
[(0, 'A'), (1, 'B'), (2, 'C')]
>>> list(zip(range(3), 'ABC', [0.0, 1.1, 2.2, 3.3]))  # ❸
[(0, 'A', 0.0), (1, 'B', 1.1), (2, 'C', 2.2)]
>>> from itertools import zip_longest  # ❹
>>> list(zip_longest(range(3), 'ABC', [0.0, 1.1, 2.2, 3.3], fillvalue=-1))
[(0, 'A', 0.0), (1, 'B', 1.1), (2, 'C', 2.2), (-1, -1, 3.3)]
```

❶ zip 函数返回一个生成器，按需生成元组。

❷ 为了输出，构建一个列表；通常，我们会迭代生成器。

❸ zip 有个奇怪的特性：当一个可迭代对象耗尽后，它不发出警告就停止。[7]

❹ itertools.zip_longest 函数的行为有所不同：使用可选的 fillvalue（默认值为 None）填充缺失的值，因此可以继续产出，直到最长的可迭代对象耗尽。

为了避免在 for 循环中手动处理索引变量，还经常使用内置的 enumerate 生成器函数。如果你不熟悉 enumerate 函数，一定要阅读“Build-in Functions”文档（https://docs.python.org/3/library/functions.html#enumerate）。内置的 zip 和 enumerate 函数，以及标准库中其他几个生成器函数在 14.9 节讨论。

[7]至少对我来说，这是奇怪的。我认为，当组合不同长度的可迭代对象时，zip 应该抛出 ValueError。

10.7　Vector类第5版：格式化

Vector 类的 __format__ 方法与 Vector2d 类的相似，但是不使用极坐标，而使用球面坐标（也叫超球面坐标），因为 Vector 类支持 *n* 个维度，而超过四维后，球体变成了“超球体”。[8] 因此，我们会把自定义的格式后缀由 'p' 变成 'h'。

[8]Wolfram Mathworld 网站中有一篇介绍超球体的文章（http://mathworld.wolfram.com/Hypersphere.html）；维基百科会把“超球体”词条重定向到“*n* 维球体”词条（http://en.wikipedia.org/wiki/N-sphere）。



9.5 节说过，扩展格式规范微语言（https://docs.python.org/3/library/string.html#formatspec）时，最好避免重用内置类型支持的格式代码。这里对微语言的扩展还会用到浮点数的格式代码 'eEfFgGn%'，而且保持原意，因此绝对要避免重用代码。整数使用的格式代码有 'bcdoxXn'，字符串使用的是 's'。在 Vector2d 类中，我选择使用 'p' 表示极坐标。使用 'h' 表示超球面坐标（hyperspherical coordinate）是个不错的选择。

例如，对四维空间（len(v) == 4）中的 Vector 对象来说，'h' 代码得到的结果是这样：<r, Φ_1, Φ_2, Φ_3>。其中，r 是模（abs(v)），余下三个数是角坐标 Φ_1、Φ_2 和 Φ_3。

下面几个示例摘自 vector_v5.py 的 doctest（参见示例 10-16），是四维球面坐标格式：

```
>>> format(Vector([-1, -1, -1, -1]), 'h')
'<2.0, 2.0943951023931957, 2.186276035465284, 3.9269908169872414>'
>>> format(Vector([2, 2, 2, 2]), '.3eh')
'<4.000e+00, 1.047e+00, 9.553e-01, 7.854e-01>'
>>> format(Vector([0, 1, 0, 0]), '0.5fh')
'<1.00000, 1.57080, 0.00000, 0.00000>'
```

在小幅改动 __format__ 方法之前，我们要定义两个辅助方法：一个是 angle(n)，用于计算某个角坐标（如 Φ_1）；另一个是 angles()，返

回由所有角坐标构成的可迭代对象。我们不会讲解其中涉及的数学原理，如果你好奇的话，可以查看维基百科中的“*n* 维球体”词条（https://en.wikipedia.org/wiki/N-sphere），那里有几个公式，我就是使用它们把 `Vector` 实例分量数组内的笛卡儿坐标转换成球面坐标的。

示例 10-16 是 vector_v5.py 脚本的完整代码，包含自 10.2 节以来实现的所有代码和本节实现的自定义格式。

示例 **10-16**　vector_v5.py：`Vector` 类最终版的 doctest 和全部代码；带标号的那几行是为了支持 `__format__` 方法而添加的代码

```
"""
A multidimensional ``Vector`` class, take 5

A ``Vector`` is built from an iterable of numbers::

    >>> Vector([3.1, 4.2])
    Vector([3.1, 4.2])
    >>> Vector((3, 4, 5))
    Vector([3.0, 4.0, 5.0])
    >>> Vector(range(10))
    Vector([0.0, 1.0, 2.0, 3.0, 4.0, ...])


Tests with two dimensions (same results as ``vector2d_v1.py``)::

    >>> v1 = Vector([3, 4])
    >>> x, y = v1
    >>> x, y
    (3.0, 4.0)
    >>> v1
    Vector([3.0, 4.0])
    >>> v1_clone = eval(repr(v1))
    >>> v1 == v1_clone
    True
    >>> print(v1)
    (3.0, 4.0)
    >>> octets = bytes(v1)
    >>> octets
    b'd\\x00\\x00\\x00\\x00\\x00\\x00\\x08@\\x00\\x00\\x00\\x00\\x00\\x00\\x1
    >>> abs(v1)
    5.0
    >>> bool(v1), bool(Vector([0, 0]))
    (True, False)
```

```
Test of ``.frombytes()`` class method:

    >>> v1_clone = Vector.frombytes(bytes(v1))
    >>> v1_clone
    Vector([3.0, 4.0])
    >>> v1 == v1_clone
    True


Tests with three dimensions::

    >>> v1 = Vector([3, 4, 5])
    >>> x, y, z = v1
    >>> x, y, z
    (3.0, 4.0, 5.0)
    >>> v1
    Vector([3.0, 4.0, 5.0])
    >>> v1_clone = eval(repr(v1))
    >>> v1 == v1_clone
    True
    >>> print(v1)
    (3.0, 4.0, 5.0)
    >>> abs(v1)  # doctest:+ELLIPSIS
    7.071067811...
    >>> bool(v1), bool(Vector([0, 0, 0]))
    (True, False)


Tests with many dimensions::

    >>> v7 = Vector(range(7))
    >>> v7
    Vector([0.0, 1.0, 2.0, 3.0, 4.0, ...])
    >>> abs(v7)  # doctest:+ELLIPSIS
    9.53939201...

Test of ``.__bytes__`` and ``.frombytes()`` methods::

    >>> v1 = Vector([3, 4, 5])
    >>> v1_clone = Vector.frombytes(bytes(v1))
    >>> v1_clone
    Vector([3.0, 4.0, 5.0])
    >>> v1 == v1_clone
    True
```

```
Tests of sequence behavior::

    >>> v1 = Vector([3, 4, 5])
    >>> len(v1)
    3
    >>> v1[0], v1[len(v1)-1], v1[-1]
    (3.0, 5.0, 5.0)


Test of slicing::

    >>> v7 = Vector(range(7))
    >>> v7[-1]
    6.0
    >>> v7[1:4]
    Vector([1.0, 2.0, 3.0])
    >>> v7[-1:]
    Vector([6.0])
    >>> v7[1,2]
    Traceback (most recent call last):
      ...
    TypeError: Vector indices must be integers


Tests of dynamic attribute access::

    >>> v7 = Vector(range(10))
    >>> v7.x
    0.0
    >>> v7.y, v7.z, v7.t
    (1.0, 2.0, 3.0)

Dynamic attribute lookup failures::

    >>> v7.k
    Traceback (most recent call last):
      ...
    AttributeError: 'Vector' object has no attribute 'k'
    >>> v3 = Vector(range(3))
    >>> v3.t
    Traceback (most recent call last):
      ...
    AttributeError: 'Vector' object has no attribute 't'
    >>> v3.spam
    Traceback (most recent call last):
      ...
    AttributeError: 'Vector' object has no attribute 'spam'
```

```
Tests of hashing::

    >>> v1 = Vector([3, 4])
    >>> v2 = Vector([3.1, 4.2])
    >>> v3 = Vector([3, 4, 5])
    >>> v6 = Vector(range(6))
    >>> hash(v1), hash(v3), hash(v6)
    (7, 2, 1)


Most hash values of non-integers vary from a 32-bit to 64-bit CPython build::

    >>> import sys
    >>> hash(v2) == (384307168202284039 if sys.maxsize > 2**32 else 357915986
    True


Tests of ``format()`` with Cartesian coordinates in 2D::

    >>> v1 = Vector([3, 4])
    >>> format(v1)
    '(3.0, 4.0)'
    >>> format(v1, '.2f')
    '(3.00, 4.00)'
    >>> format(v1, '.3e')
    '(3.000e+00, 4.000e+00)'


Tests of ``format()`` with Cartesian coordinates in 3D and 7D::

    >>> v3 = Vector([3, 4, 5])
    >>> format(v3)
    '(3.0, 4.0, 5.0)'
    >>> format(Vector(range(7)))
    '(0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0)'


Tests of ``format()`` with spherical coordinates in 2D, 3D and 4D::

    >>> format(Vector([1, 1]), 'h')  # doctest:+ELLIPSIS
    '<1.414213..., 0.785398...>'
    >>> format(Vector([1, 1]), '.3eh')
    '<1.414e+00, 7.854e-01>'
    >>> format(Vector([1, 1]), '0.5fh')
    '<1.41421, 0.78540>'
    >>> format(Vector([1, 1, 1]), 'h')  # doctest:+ELLIPSIS
    '<1.73205..., 0.95531..., 0.78539...>'
```

```
    >>> format(Vector([2, 2, 2]), '.3eh')
    '<3.464e+00, 9.553e-01, 7.854e-01>'
    >>> format(Vector([0, 0, 0]), '0.5fh')
    '<0.00000, 0.00000, 0.00000>'
    >>> format(Vector([-1, -1, -1, -1]), 'h')  # doctest:+ELLIPSIS
    '<2.0, 2.09439..., 2.18627..., 3.92699...>'
    >>> format(Vector([2, 2, 2, 2]), '.3eh')
    '<4.000e+00, 1.047e+00, 9.553e-01, 7.854e-01>'
    >>> format(Vector([0, 1, 0, 0]), '0.5fh')
    '<1.00000, 1.57080, 0.00000, 0.00000>'
"""

from array import array
import reprlib
import math
import numbers
import functools
import operator
import itertools  ❶


class Vector:
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components)

    def __iter__(self):
        return iter(self._components)

    def __repr__(self):
        components = reprlib.repr(self._components)
        components = components[components.find('['):-1]
        return 'Vector({})'.format(components)

    def __str__(self):
        return str(tuple(self))

    def __bytes__(self):
        return (bytes([ord(self.typecode)]) +
                bytes(self._components))

    def __eq__(self, other):
        return (len(self) == len(other) and
                all(a == b for a, b in zip(self, other)))

    def __hash__(self):
        hashes = (hash(x) for x in self)
```

```
        return functools.reduce(operator.xor, hashes, 0)

    def __abs__(self):
        return math.sqrt(sum(x * x for x in self))

    def __bool__(self):
        return bool(abs(self))

    def __len__(self):
        return len(self._components)

    def __getitem__(self, index):
        cls = type(self)
        if isinstance(index, slice):
            return cls(self._components[index])
        elif isinstance(index, numbers.Integral):
            return self._components[index]
        else:
            msg = '{.__name__} indices must be integers'
            raise TypeError(msg.format(cls))

    shortcut_names = 'xyzt'

    def __getattr__(self, name):
        cls = type(self)
        if len(name) == 1:
            pos = cls.shortcut_names.find(name)
            if 0 <= pos < len(self._components):
                return self._components[pos]
        msg = '{.__name__!r} object has no attribute {!r}'
        raise AttributeError(msg.format(cls, name))

    def angle(self, n):  ❷
        r = math.sqrt(sum(x * x for x in self[n:]))
        a = math.atan2(r, self[n-1])
        if (n == len(self) - 1) and (self[-1] < 0):
            return math.pi * 2 - a
        else:
            return a

    def angles(self):  ❸
        return (self.angle(n) for n in range(1, len(self)))

    def __format__(self, fmt_spec=''):
        if fmt_spec.endswith('h'):  # 超球面坐标
            fmt_spec = fmt_spec[:-1]
            coords = itertools.chain([abs(self)],
                                     self.angles())  ❹
```

```
            outer_fmt = '<{}>'  ❺
        else:
            coords = self
            outer_fmt = '({})'  ❻
        components = (format(c, fmt_spec) for c in coords)  ❼
        return outer_fmt.format(', '.join(components))  ❽

    @classmethod
    def frombytes(cls, octets):
        typecode = chr(octets[0])
        memv = memoryview(octets[1:]).cast(typecode)
        return cls(memv)
```

❶ 为了在 `__format__` 方法中使用 `chain` 函数，导入 `itertools` 模块。

❷ 使用“*n* 维球体”词条（http://en.wikipedia.org/wiki/N-sphere）中的公式计算某个角坐标。

❸ 创建生成器表达式，按需计算所有角坐标。

❹ 使用 `itertools.chain` 函数生成生成器表达式，无缝迭代向量的模和各个角坐标。

❺ 配置使用尖括号显示球面坐标。

❻ 配置使用圆括号显示笛卡儿坐标。

❼ 创建生成器表达式，按需格式化各个坐标元素。

❽ 把以逗号分隔的格式化分量插入尖括号或圆括号。

> 我们在 `__format__`、`angle` 和 `angles` 中大量使用了生成器表达式，不过我们的目的是让 `Vector` 类的 `__format__` 方法与 `Vector2d` 类处在同一水平上。第 14 章讨论生成器时会使用 `Vector` 类中的部分代码举例，然后详细说明生成器的技巧。

本章的任务到此结束。第 13 章会改进 `Vector` 类，让它支持中缀运算符。本章的目的是探讨如何编写集合类广泛使用的几个特殊方法。

10.8　本章小结

本章所举的 Vector 示例故意与 Vector2d 兼容，不过二者的构造方法签名不同，Vector 类的构造方法接受一个可迭代的对象，这与内置的序列类型一样。Vector 的行为之所以像序列，是因为它实现了 __getitem__ 和 __len__ 方法；借此，我们讨论了协议，这是鸭子类型语言使用的非正式接口。

然后，我们说明了 my_seq[a:b:c] 句法背后的工作原理：创建 slice(a, b, c) 对象，交给 __getitem__ 方法处理。了解这一点之后，我们让 Vector 正确处理切片，像符合 Python 风格的序列那样返回新的 Vector 实例。

接下来，我们为 Vector 实例的头几个分量提供了只读访问功能，使用 my_vec.x 这样的表示法。这一点通过 __getattr__ 方法实现。实现这一功能之后，用户会想通过 my_vec.x = 7 这样的写法为头几个分量赋值——这是一个潜在的缺陷。为了解决这个问题，我们又实现了 __setattr__ 方法，通过它禁止为单字母属性赋值。大多数时候，如果定义了 __getattr__ 方法，那么也要定义 __setattr__ 方法，这样才能避免行为不一致。

实现 __hash__ 方法特别适合使用 functools.reduce 函数，因为我们要把异或运算符 ^ 依次应用到各个分量的散列值上，生成整个向量的聚合散列值。在 __hash__ 方法中使用 reduce 函数之后，我们又使用内置的归约函数 all 实现了效率更高的 __eq__ 方法。

Vector 类的最后一项改进是在 Vector2d 的基础上重新实现 __format__ 方法，这一次，除了支持笛卡儿坐标，我们还支持了球面坐标。为了定义 __format__ 方法及其辅助方法，我们用到了很多数学知识和几个生成器，但这些是实现细节。第 14 章会再次讨论生成器。最后一节的目的是支持自定义格式，从而兑现承诺，让 Vector 与 Vector2d 兼容，此外还能做更多的事情。

与第 9 章一样，我们经常分析 Python 标准对象的行为，然后进行模仿，让 Vector 的行为符合 Python 风格。

第 13 章将为 `Vector` 实现几个中缀运算符。第 13 章使用的数学知识比 `angle()` 方法用到的简单多了，但是通过了解 Python 中缀运算符的工作方式，我们对面向对象设计的认识将更进一步。讨论运算符重载之前，我们将先定义一个类，说明如何使用接口和继承组织多个类——这是第 11 章和第 12 章的话题。

10.9　延伸阅读

`Vector` 类中的大多数特殊方法在第 9 章定义的 `Vector2d` 类中也有，因此前一章给出的延伸阅读材料同样适合本章。

强大的高阶函数 `reduce` 也叫合拢、累计、聚合、压缩和注入。更多信息参见维基百科中的“Fold (higher-order function)”词条（https://en.wikipedia.org/wiki/Fold_(higher-order_function)）。这篇文章展示了高阶函数的用途，着重说明了具有递归数据结构的函数式语言。这篇文章中还有一个表格，列出了很多编程语言中起合拢作用的函数。

杂谈

把协议当作非正式的接口

协议不是 Python 发明的。Smalltalk 团队，也就是“面向对象”的发明者，使用“协议”这个词表示现在我们称之为接口的特性。某些 Smalltalk 编程环境允许程序员把一组方法标记为协议，但这只不过是一种文档，用于辅助导航，语言不对其施加特定措施。因此，向熟悉正式（而且编译器会施加措施）接口的人解释“协议”时，我会简单地说它是“非正式的接口”。

动态类型语言中的既定协议会自然进化。所谓动态类型是指在运行时检查类型，因为方法签名和变量没有静态类型信息。Ruby 是一门重要的面向对象动态类型语言，它也使用协议。

在 Python 文档中，如果看到“文件类对象”这样的表述，通常说的就是协议。这是一种简短的说法，意思是：“行为基本与文件一致，实现了部分文件接口，满足上下文相关需求的东西。”

你可能觉得只实现协议的一部分不够严谨，但是这样做的优点是简单。“Data Model”一章的 3.3 节（https://docs.python.org/3/reference/datamodel.html#special-method-names）建议：

> 模仿内置类型实现类时，记住一点：模仿的程度对建模的对象

来说合理即可。例如，有些序列可能只需要获取单个元素，而不必提取切片。

——Python 语言参考手册中“Data Model”一章

不要为了满足过度设计的接口契约和让编译器开心，而去实现不需要的方法，我们要遵守 KISS 原则（http://en.wikipedia.org/wiki/KISS_principle）。

第 11 章还会讨论协议和接口，这正是那一章的主要话题。

鸭子类型的起源

我相信 Ruby 社区在“鸭子类型”这个术语的推广过程中起了主要作用，因为他们向大量 Java 使用者宣扬了这个说法。但是，在 Ruby 或 Python 流行起来之前，Python 就使用这个术语了。根据维基百科，在面向对象编程中较早使用鸭子作比喻的人是 Alex Martelli，在他于 2000 年 7 月 26 日发到 Python-list 中的一篇文章里：“polymorphism (was Re: Type checking in python?)”（https://mail.python.org/pipermail/python-list/2000-July/046184.html）。本章开头引用的那句话就出自那篇文章。如果你想知道“鸭子类型”这个术语的真正起源，以及很多编程语言对这个面向对象概念的运用，请阅读维基百科中的“Duck typing”词条（http://en.wikipedia.org/wiki/Duck_typing）。

安全的 **`__format__`** 方法，增强可用性

实现 `__format__` 方法时，我们没有采取措施防范 `Vector` 实例拥有大量分量，不过在 `__repr__` 方法中我们使用 `reprlib` 做了预防。这是因为 `repr()` 函数用于调试和记录日志，所以必须生成可用的输出；而 `__format__` 方法用于向最终用户显示输出，他们大概想看到整个 `Vector`。如果你觉得这样做危险，可以再为格式规范微语言实现一个扩展。

如果是我，我会这么做：默认情况下，格式化的 `Vector` 实例显示有限个分量，比如说 30 个。如果元素数量超过上限，默认的行为是像 `reprlib` 那样，截断超出的部分，使用 `...` 表示。然而，如果格式说明符后面有特殊的 `*` 代码（意思是“全部”），那么就不限

制显示的元素数量。因此，用户在不知情的情况下不会被特别长的输出吓到。如果默认的上限碍事，那么 ... 的存在对用户是个提醒，用户研究文档后会发现 * 格式代码。

如果你实现了，请向本书的 GitHub 仓库（https://github.com/fluentpython/example-code）发一个拉取请求。

寻找符合 **Python** 风格的求和方式

就像“什么是美”没有确切的答案一样，“什么是 Python 风格”也没有标准答案。如果回答“地道的 Python”（我通常会这样说），不能让人 100% 满意，因为对你来说是“地道的”，在我看来却可能不是。但我可以肯定的是，“地道”并不是指使用最鲜为人知的语言特性。

Python-list（https://mail.python.org/mailman/listinfo/python-list）中有一篇发表于 2003 年 4 月的话题，题为“Pythonic Way to Sum n-th List Element?”（https://mail.python.org/pipermail/python-list/2003-April/218568.html）。这个话题与本章讨论的 reduce 函数有关。

该话题的发起人 Guy Middleton 说他不喜欢使用 lambda 表达式，问下面这个方案有没有办法改进：[9]

```
>>> my_list = [[1, 2, 3], [40, 50, 60], [9, 8, 7]]
>>> import functools
>>> functools.reduce(lambda a, b: a+b, [sub[1] for sub in my_list])
60
```

这段代码有很多习惯用法：lambda、reduce 和列表推导。最终，这可能会变成人气竞赛，因为它冒犯了讨厌 lambda 的人和看不上列表推导的人——这两种人都很多。

如果使用 lambda，或许就不应该使用列表推导——过滤除外，但这不是过滤。

下面是我给出的方案，这能讨得 lambda 拥护者的欢心：

```
>>> functools.reduce(lambda a, b: a + b[1], my_list, 0)
60
```

我没有参与那个话题，而且我不会在真实的代码中使用上述方案，因为我非常不喜欢 `lambda` 表达式。这里只是为了举例说明不使用列表推导怎么做。

第一个答案是 Fernando Perez 给出的，他是 IPython 的创建者，他的答案强调了 NumPy 支持 n 维数组和 n 维切片：

```
>>> import numpy as np
>>> my_array = np.array(my_list)
>>> np.sum(my_array[:, 1])
60
```

我觉得 Perez 的方案很棒，不过 Guy Middleton 推崇 Paul Rubin 和 Skip Montanaro 给出的下述方案：

```
>>> import operator
>>> functools.reduce(operator.add, [sub[1] for sub in my_list], 0)
60
```

随后，Evan Simpson 问道："这样做有什么错？"

```
>>> total = 0
>>> for sub in my_list:
...     total += sub[1]
>>> total
60
```

许多人都觉得这也很符合 Python 风格。Alex Martelli 甚至说，Guido 或许就会这么做。

我喜欢 Evan Simpson 的代码，不过也喜欢 David Eppstein 对此给出的评论：

> 如果你想计算列表中各个元素的和，写出的代码应该看起来像是在"计算元素之和"，而不是"迭代元素，维护一个变量 t，再

> 执行一系列求和操作”。如果不能站在一定高度上表明意图，让语言去关注低层操作，那么要高级语言干嘛？

之后 Alex Martelli 又建议：

> 求和操作经常需要，我不介意 Python 提供一个这样的内置函数。但是，在我看来，“reduce(operator.add, ...”不是好方法（作为一名 APL 老程序员和 FP 语言的爱好者，我应该喜欢，但是我并不喜欢）。

随后，Alex 建议提供并实现了 `sum()` 函数。这次讨论之后三个月，Python 2.3 就内置了这个函数。因此，Alex 喜欢的句法变成了标准：

```
>>> sum([sub[1] for sub in my_list])
60
```

下一年年末（2004 年 11 月），Python 2.4 发布了，这一版引入了生成器表达式。因此，在我看来，Guy Middleton 那个问题目前最符合 Python 风格的答案是：

```
>>> sum(sub[1] for sub in my_list)
60
```

这样写不仅比使用 `reduce` 函数更易阅读，而且还能避免空序列导致的陷阱：`sum([])` 的结果是 `0`，就这么简单。

在这次讨论中，Alex Martelli 指出，Python 2 内置的 `reduce` 函数成事不足败事有余，因为它推荐的地道编程方式难以理解。他的观点最有说服力：Python 3 把 `reduce` 函数移到 `functools` 模块中了。

当然，`functools.reduce` 函数仍有它的作用。实现 `Vector.__hash__` 方法时我就用了它，我觉得我的实现方式算得上符合 Python 风格。

> [9]为了在此展示，我稍微修改了这段代码，因为在 2003 年，`reduce` 是内置函数，而在 Python 3 中要导入。此外，我把 `x` 和 `y` 换成了 `my_list` 和 `sub`（表示子串）。

第 11 章　接口：从协议到抽象基类

抽象类表示接口。[1]

——Bjarne Stroustrup
C++ 之父

[1]Bjarne Stroustrup, *The Design and Evolution of C++* (Addison-Wesley, 1994), p. 278.

本章讨论的话题是接口：从鸭子类型的代表特征动态协议，到使接口更明确、能验证实现是否符合规定的抽象基类（Abstract Base Class，ABC）。

如果用过 Java、C#或类似的语言，你会觉得鸭子类型的非正式协议很新奇。但是对长时间使用 Python 或 Ruby 的程序员来说，这是接口的“常规”方式，新知识是抽象基类的严格规定和类型检查。Python 语言诞生 15 年后，Python 2.6 才引入抽象基类。

本章先说明 Python 社区以往对接口的不严谨理解：部分实现接口通常被认为是可接受的。我们将通过几个示例强调鸭子类型的动态本性，从而澄清这一点。

接着，我邀请 Alex Martelli 写了一篇短文，对抽象基类做了介绍，还为 Python 编程的一个新趋势下了定义。本章余下的内容专门讲解抽象基类。首先，本章说明抽象基类的常见用途：实现接口时作为超类使用。然后，说明抽象基类如何检查具体子类是否符合接口定义，以及如何使用注册机制声明一个类实现了某个接口，而不进行子类化操作。最后，说明如何让抽象基类自动“识别”任何符合接口的类——不进行子类化或注册。

我们将实现一个新抽象基类，看看它的运作方式。但是，我和 Alex Martelli 都不建议你自己编写抽象基类，因为很容易过度设计。

抽象基类与描述符和元类一样，是用于构建框架的工具。因此，只有少数 Python 开发者编写的抽象基类不会对用户施加不必要的限制，让他们做无用功。

下面我们从 Python 风格的角度探讨接口。

11.1　Python文化中的接口和协议

引入抽象基类之前，Python 就已经非常成功了，即便现在也很少有代码使用抽象基类。第 1 章就已经讨论了鸭子类型和协议。在 10.3 节，我们把协议定义为非正式的接口，是让 Python 这种动态类型语言实现多态的方式。

接口在动态类型语言中是怎么运作的呢？首先，基本的事实是，Python 语言没有 `interface` 关键字，而且除了抽象基类，每个类都有接口：类实现或继承的公开属性（方法或数据属性），包括特殊方法，如 `__getitem__` 或 `__add__`。

按照定义，受保护的属性和私有属性不在接口中：即便“受保护的”属性也只是采用命名约定实现的（单个前导下划线）；私有属性可以轻松地访问（参见 9.7 节），原因也是如此。不要违背这些约定。

另一方面，不要觉得把公开数据属性放入对象的接口中不妥，因为如果需要，总能实现读值方法和设值方法，把数据属性变成特性，使用 `obj.attr` 句法的客户代码不会受到影响。 `Vector2d` 类就是这么做的，示例 11-1 是 `Vector2d` 类的第一版，`x` 和 `y` 是公开属性。

示例 **11-1**　vector2d_v0.py：`x` 和 `y` 是公开数据属性（代码与示例 9-2 相同）

```
class Vector2d:
    typecode = 'd'

    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    def __iter__(self):
        return (i for i in (self.x, self.y))

    # 下面是其他方法（这个代码清单将其省略了）
```

在示例 9-7 中，我们把 `x` 和 `y` 变成了只读特性（见示例 11-2）。这是一

项重大重构，但是 `Vector2d` 的接口基本没变：用户仍能读取 `my_vector.x` 和 `my_vector.y`。

示例 **11-2**　vector2d_v3.py：使用特性实现 `x` 和 `y`（完整的代码清单参见示例 9-9）

```
class Vector2d:
    typecode = 'd'

    def __init__(self, x, y):
        self.__x = float(x)
        self.__y = float(y)

    @property
    def x(self):
        return self.__x

    @property
    def y(self):
        return self.__y

    def __iter__(self):
        return (i for i in (self.x, self.y))

    # 下面是其他方法（这个代码清单将其省略了）
```

关于接口，这里有个实用的补充定义：对象公开方法的子集，让对象在系统中扮演特定的角色。Python 文档中的“文件类对象”或“可迭代对象”就是这个意思，这种说法指的不是特定的类。接口是实现特定角色的方法集合，这样理解正是 Smalltalk 程序员所说的协议，其他动态语言社区都借鉴了这个术语。协议与继承没有关系。一个类可能会实现多个接口，从而让实例扮演多个角色。

协议是接口，但不是正式的（只由文档和约定定义），因此协议不能像正式接口那样施加限制（本章后面会说明抽象基类对接口一致性的强制）。一个类可能只实现部分接口，这是允许的。有时，某些 API 只要求“文件类对象”返回字节序列的 `.read()` 方法。在特定的上下文中可能需要其他文件操作方法，也可能不需要。

写作本书时，Python 3 中 `memoryview` 的文档

（https://docs.python.org/3/library/stdtypes.html#typememoryview）说，它能处理“支持缓冲协议的对象”，不过缓冲协议的文档是针对 C API 的。`bytearray` 的构造方法（https://docs.python.org/3/library/functions.html#bytearray）接受“一个符合缓冲接口的对象”。如今，文档正在改变用词，使用“字节序列类对象”这样更加友好的表述。[2] 我指出这一点是为了强调，对 Python 程序员来说，“X 类对象”“X 协议”和“X 接口”都是一个意思。

> [2]其实，Issue 16518:“add buffer protocol to glossary”（http://bugs.python.org/issue16518）做的就是这种修改，把很多“支持缓冲协议 / 接口 /API 的对象”改成了“字节序列类对象”；“Other mentions of the buffer protocol”（http://bugs.python.org/issue22581）也是如此。

序列协议是 Python 最基础的协议之一。即便对象只实现了那个协议最基本的一部分，解释器也会负责任地处理，如下一节所示。

11.2　Python喜欢序列

Python 数据模型的哲学是尽量支持基本协议。对序列来说，即便是最简单的实现，Python 也会力求做到最好。

图 11-1 展示了定义为抽象基类的 `Sequence` 正式接口。

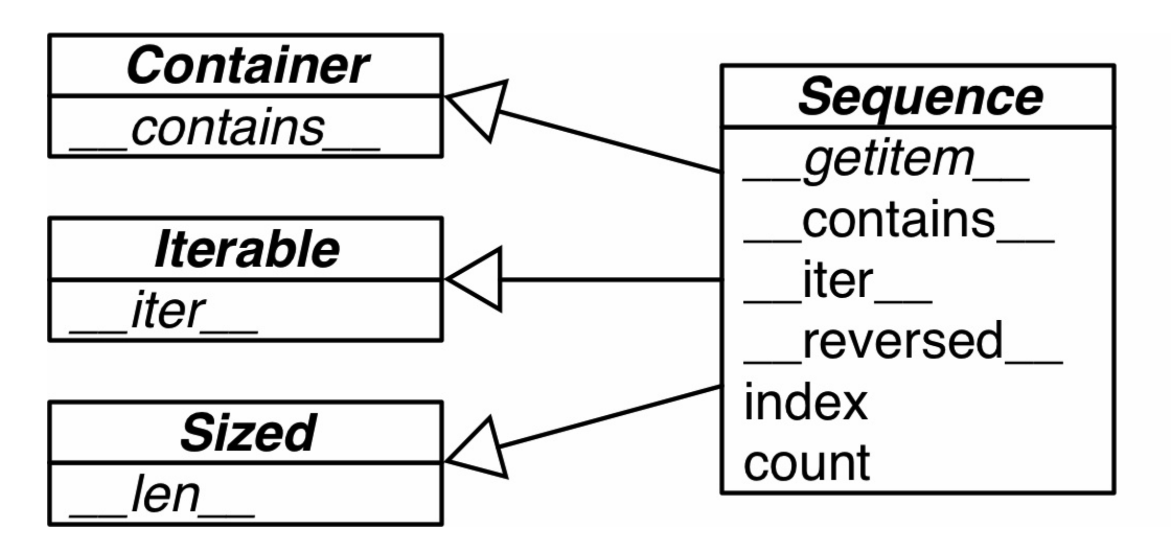


图 **11-1**：**`Sequence`** 抽象基类和 **`collections.abc`** 中相关抽象类的 **UML** 类图，箭头由子类指向超类，以斜体显示的是抽象方法

现在，看看示例 11-3 中的 `Foo` 类。它没有继承 `abc.Sequence`，而且只实现了序列协议的一个方法：`__getitem__`（没有实现 `__len__` 方法）。

示例 **11-3**　定义 `__getitem__` 方法，只实现序列协议的一部分，这样足够访问元素、迭代和使用 `in` 运算符了

```
>>> class Foo:
...     def __getitem__(self, pos):
...         return range(0, 30, 10)[pos]
...
>>> f = Foo()
>>> f[1]
10
>>> for i in f: print(i)
```

```
...
0
10
20
>>> 20 in f
True
>>> 15 in f
False
```

虽然没有 `__iter__` 方法，但是 Foo 实例是可迭代的对象，因为发现有 `__getitem__` 方法时，Python 会调用它，传入从 0 开始的整数索引，尝试迭代对象（这是一种后备机制）。尽管没有实现 `__contains__` 方法，但是 Python 足够智能，能迭代 Foo 实例，因此也能使用 in 运算符：Python 会做全面检查，看看有没有指定的元素。

综上，鉴于序列协议的重要性，如果没有 `__iter__` 和 `__contains__` 方法，Python 会调用 `__getitem__` 方法，设法让迭代和 in 运算符可用。

第 1 章定义的 FrenchDeck 类也没有继承 abc.Sequence，但是实现了序列协议的两个方法：`__getitem__` 和 `__len__`。如示例 11-4 所示。

示例 **11-4**　实现序列协议的 FrenchDeck 类（代码与示例 1-1 相同）

```
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                                        for rank in self.ranks]

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, position):
        return self._cards[position]
```

第 1 章那些示例之所以能用，大部分是由于 Python 会特殊对待看起来像是序列的对象。Python 中的迭代是鸭子类型的一种极端形式：为了迭代对象，解释器会尝试调用两个不同的方法。

下面再分析一个示例，着重强调协议的动态本性。

11.3　使用猴子补丁在运行时实现协议

示例 11-4 中的 `FrenchDeck` 类有个重大缺陷：无法洗牌。几年前，第一次编写 `FrenchDeck` 示例时，我实现了 `shuffle` 方法。后来，我对 Python 风格有了深刻理解，我发现如果 `FrenchDeck` 实例的行为像序列，那么它就不需要 `shuffle` 方法，因为已经有 `random.shuffle` 函数可用，文档中说它的作用是“就地打乱序列 *x*”（https://docs.python.org/3/library/random.html#random.shuffle）。

> 如果遵守既定协议，很有可能增加利用现有的标准库和第三方代码的可能性，这得益于鸭子类型。

标准库中的 `random.shuffle` 函数用法如下：

```
>>> from random import shuffle
>>> l = list(range(10))
>>> shuffle(l)
>>> l
[5, 2, 9, 7, 8, 3, 1, 4, 0, 6]
```

然而，如果尝试打乱 `FrenchDeck` 实例，会出现异常，如示例 11-5 所示。

示例 **11-5**　`random.shuffle` 函数不能打乱 `FrenchDeck` 实例

```
>>> from random import shuffle
>>> from frenchdeck import FrenchDeck
>>> deck = FrenchDeck()
>>> shuffle(deck)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../python3.3/random.py", line 265, in shuffle
    x[i], x[j] = x[j], x[i]
TypeError: 'FrenchDeck' object does not support item assignment
```

错误消息相当明确，“`'FrenchDeck'` object does not support item assignment”（`'FrenchDeck'` 对象不支持为元素赋值）。这个问题的原

因是，shuffle 函数要调换集合中元素的位置，而 FrenchDeck 只实现了不可变的序列协议。可变的序列还必须提供 __setitem__ 方法。

Python 是动态语言，因此我们可以在运行时修正这个问题，甚至还可以在交互式控制台中，修正方法如示例 11-6 所示。

示例 **11-6** 为FrenchDeck 打猴子补丁，把它变成可变的，让 random.shuffle 函数能处理（接续示例 11-5）

```
>>> def set_card(deck, position, card): ❶
...     deck._cards[position] = card
...
>>> FrenchDeck.__setitem__ = set_card ❷
>>> shuffle(deck) ❸
>>> deck[:5]
[Card(rank='3', suit='hearts'), Card(rank='4', suit='diamonds'), Card(rank='4
suit='clubs'), Card(rank='7', suit='hearts'), Card(rank='9', suit='spades')]
```

❶ 定义一个函数，它的参数为 deck、position 和 card。

❷ 把那个函数赋值给 FrenchDeck 类的 __setitem__ 属性。

❸ 现在可以打乱 deck 了，因为 FrenchDeck 实现了可变序列协议所需的方法。

特殊方法 __setitem__ 的签名在 Python 语言参考手册的“3.3.6. Emulating container types”（https://docs.python.org/3/reference/datamodel.html#emulating-container-types）中定义。语言参考中使用的参数是 self、key 和 value，而这里使用的是 deck、position 和 card。这么做是为了告诉你，每个 Python 方法说到底都是普通函数，把第一个参数命名为 self 只是一种约定。在控制台会话中使用那几个参数没问题，不过在 Python 源码文件中最好按照文档那样使用 self、key 和 value。

这里的关键是，set_card 函数要知道 deck 对象有一个名为 _cards 的属性，而且 _cards 的值必须是可变序列。然后，我们把 set_card 函数赋值给特殊方法 __setitem__，从而把它依附到 FrenchDeck 类上。这种技术叫猴子补丁：在运行时修改类或模块，而不改动源码。猴子补丁很强大，但是打补丁的代码与要打补丁的程序耦合十分紧密，而

且往往要处理隐藏和没有文档的部分。

除了举例说明猴子补丁之外，示例 11-6 还强调了协议是动态的：`random.shuffle` 函数不关心参数的类型，只要那个对象实现了部分可变序列协议即可。即便对象一开始没有所需的方法也没关系，后来再提供也行。

目前，本章讨论的主题是“鸭子类型”：对象的类型无关紧要，只要实现了特定的协议即可。

前面给出的抽象基类图表是为了展示协议与抽象基类的文档中所说的接口之间的关系，但是目前为止还没有真正继承抽象基类。

在接下来的几节中，我们将直接使用抽象基类，而不只将其当作文档。

11.4　Alex Martelli的水禽

介绍完 Python 常规的协议风格接口后，下面讨论抽象基类。不过在分析示例和细节之前，我们要看 Alex Martelli 写的一篇短文。这篇短文说明了 Python 为什么引入抽象基类。

> 非常感谢 Alex Martelli。本书引用最多的就是他说的话，后来他变成了本书的技术编辑之一。他的见解已经非常宝贵了，现在又愿意撰写这篇短文。Python 社区有他的存在真是幸运。接下来交给你了，Alex！

水禽和抽象基类

Alex Martelli 撰

维基百科（http://en.wikipedia.org/wiki/Duck_typing#History）说是我协助传播了“鸭子类型”这种言简意赅的说法（即忽略对象的真正类型，转而关注对象有没有实现所需的方法、签名和语义）。

对 Python 来说，这基本上是指避免使用 `isinstance` 检查对象的类型（更别提 `type(foo) is bar` 这种更糟的检查方式了，这样做没有任何好处，甚至禁止最简单的继承方式）。

总的来说，鸭子类型在很多情况下十分有用；但是在其他情况下，随着发展，通常有更好的方式。事情是这样的……

近代，属和种（包括但不限于水禽所属的鸭科）基本上是根据表型系统学（phenetics）分类的。表征学关注的是形态和举止的相似性……主要是表型系统学特征。因此使用“鸭子类型”比喻是贴切的。

然而，平行进化往往会导致不相关的种产生相似的特征，形态和举止方面都是如此，但是生态位的相似性是偶然的，不同的种仍属不

同的生态位。编程语言中也有这种“偶然的相似性”，比如说下述经典的面向对象编程示例：

```
class Artist:
    def draw(self): ...

class Gunslinger:
    def draw(self): ...

class Lottery:
    def draw(self): ...
```

显然，只因为 `x` 和 `y` 两个对象刚好都有一个名为 `draw` 的方法，而且调用时不用传入参数，即 `x.draw()` 和 `y.draw()`，远远不能确保二者可以相互调用，或者具有相同的抽象。也就是说，从这样的调用中不能推导出语义相似性。相反，我们需要一位渊博的程序员主动把这种等价维持在一定层次上。

生物（和其他学科）遇到的这个问题，迫切需要（从很多方面来说，是催生）表征学之外的分类方式解决，即支序系统学（cladistics）。这种分类学主要根据从共同祖先那里继承的特征分类，而不是单独进化的特征。（近些年，DNA 测序变得便宜又快，这使支序学的实用地位变得更高。）

例如，草雁（以前认为与其他鹅类比较相似）和麻鸭（以前认为与其他鸭类比较相似）现在被分到 Tadornidae 亚科（表明二者的相似性比鸭科中其他动物高，因为它们的共同祖先比较接近）。此外，DNA 分析表明，白翅木鸭与美洲家鸭（属于麻鸭）不是很像，至少没有形态和举止看起来那么像，因此把木鸭单独分成了一属，完全不在 Tadornidae 亚科中。

知道这些有什么用呢？视情况而定！比如，逮到一只水禽后，决定如何烹制才最美味时，显著的特征（不是全部，例如一身羽毛并不重要）主要是口感和风味（过时的表征学），这比支序学重要得多。但在其他方面，如对不同病原体的抗性（圈养水禽还是放养），DNA 接近性的作用就大多了……

因此，参照水禽的分类学演化，我建议在鸭子类型的基础上增加

白鹅类型（goose typing）。

白鹅类型指，只要 cls 是抽象基类，即 cls 的元类是 abc.ABCMeta，就可以使用 isinstance(obj, cls)。

collections.abc 中有很多有用的抽象类（Python 标准库的 numbers 模块中还有一些）。[3]

与具体类相比，抽象基类有很多理论上的优点（例如，参阅 Scott Meyer 写的《More Effective C++：35 个改善编程与设计的有效方法（中文版）》的“条款 33：将非尾端类设计为抽象类”，英文版见 http://ptgmedia.pearsoncmg.com/images/020163371x/items/item33.html），Python 的抽象基类还有一个重要的实用优势：可以使用 register 类方法在终端用户的代码中把某个类“声明”为一个抽象基类的“虚拟”子类（为此，被注册的类必须满足抽象基类对方法名称和签名的要求，最重要的是要满足底层语义契约；但是，开发那个类时不用了解抽象基类，更不用继承抽象基类）。这大大地打破了严格的强耦合，与面向对象编程人员掌握的知识有很大出入，因此使用继承时要小心。

有时，为了让抽象基类识别子类，甚至不用注册。

其实，抽象基类的本质就是几个特殊方法。例如：

```
>>> class Struggle:
...     def __len__(self): return 23
...
>>> from collections import abc
>>> isinstance(Struggle(), abc.Sized)
True
```

可以看出，无需注册，abc.Sized 也能把 Struggle 识别为自己的子类，只要实现了特殊方法 __len__ 即可（要使用正确的句法和语义实现，前者要求没有参数，后者要求返回一个非负整数，指明对象的长度；如果不使用规定的句法和语义实现特殊方法，如 __len__，会导致非常严重的问题）。

最后我想说的是：如果实现的类体现了

numbers、collections.abc 或其他框架中抽象基类的概念，要么继承相应的抽象基类（必要时），要么把类注册到相应的抽象基类中。开始开发程序时，不要使用提供注册功能的库或框架，要自己动手注册；如果必须检查参数的类型（这是最常见的），例如检查是不是“序列”，那就这样做：

```
isinstance(the_arg, collections.abc.Sequence)
```

此外，不要在生产代码中定义抽象基类（或元类）……如果你很想这样做，我打赌可能是因为你想“找茬”，刚拿到新工具的人都有大干一场的冲动。如果你能避开这些深奥的概念，你（以及未来的代码维护者）的生活将更愉快，因为代码会变得简洁明了。再会！

> [3]当然，你还可以自己定义抽象基类，但是我不建议高级 Python 程序员之外的人这么做；同样，我也不建议你自己定义元类……我说的“高级 Python 程序员”是指对 Python 语言的一招一式都了如指掌，即便对这类人来说，抽象基类和元类也不是常用工具。如此“深层次的元编程”，如果可以这么讲的话，适合框架的作者使用，这样便于众多不同的开发团队独立扩展框架……真正需要这么做的“高级 Python 程序员”不超过 1%。——Alex Martelli

除了提出“白鹅类型”之外，Alex 还指出，继承抽象基类很简单，只需要实现所需的方法，这样也能明确表明开发者的意图。这一意图还能通过注册虚拟子类来实现。

此外，使用 isinstance 和 issubclass 测试抽象基类更为人接受。过去，这两个函数用来测试鸭子类型，但用于抽象基类会更灵活。毕竟，如果某个组件没有继承抽象基类，事后还可以注册，让显式类型检查通过。

然而，即便是抽象基类，也不能滥用 isinstance 检查，用得多了可能导致代码异味，即表明面向对象设计得不好。在一连串 if/elif/elif 中使用 isinstance 做检查，然后根据对象的类型执行不同的操作，通常是不好的做法；此时应该使用多态，即采用一定的方式定义类，让解释器把调用分派给正确的方法，而不使用 if/elif/elif 块硬编码分派逻辑。



具体使用时，上述建议有一个常见的例外：有些 Python API

接受一个字符串或字符串序列；如果只有一个字符串，可以把它放到列表中，从而简化处理。因为字符串是序列类型，所以为了把它和其他不可变序列区分开，最简单的方式是使用 `isinstance(x, str)` 检查。[4]

> [4]可惜，在 Python 3.4 中没有能把字符串和元组或其他不可变序列区分开的抽象基类，因此必须测试 `str`。在 Python 2 中，`basestr` 类型可以协助这样的测试。`basestr` 不是抽象基类，但它是 `str` 和 `unicode` 的超类；然而，Python 3 把 `basestr` 去掉了。奇怪的是，Python 3 中有个 `collections.abc.ByteString` 类型，但是它只能检测 `bytes` 和 `bytearray` 类型。

另一方面，如果必须强制执行 API 契约，通常可以使用 `isinstance` 检查抽象基类。“老兄，如果你想调用我，必须实现这个”，正如本书技术审校 Lennart Regebro 所说的。这对采用插入式架构的系统来说特别有用。在框架之外，鸭子类型通常比类型检查更简单，也更灵活。

例如，本书有几个示例要使用序列，把它当成列表处理。我没有检查参数的类型是不是 `list`，而是直接接受参数，立即使用它构建一个列表。这样，我就可以接受任何可迭代对象；如果参数不是可迭代对象，调用立即失败，并且提供非常清晰的错误消息。本章后面示例 11-13 中的代码就是这么做的。当然，如果序列太长或者需要就地修改序列而导致无法复制参数，就不能采用这种方式；此时，使用 `isinstance(x, abc.MutableSequence)` 更好。如果可以接受任何可迭代对象，也可以调用 `iter(x)` 函数获得一个迭代器，详情参见 14.1.1 节。

模仿 `collections.namedtuple`（https://docs.python.org/3/library/collections.h 处理 `field_names` 参数的方式也是一例：`field_names` 的值可以是单个字符串，以空格或逗号分隔标识符，也可以是一个标识符序列。此时可能想使用 `isinstance`，但我会使用鸭子类型，如示例 11-7 所示。[5]

> [5]这段代码摘自示例 21-2。

示例 **11-7**　使用鸭子类型处理单个字符串或由字符串组成的可迭代对象

```
try: ❶
    field_names = field_names.replace(',', ' ').split() ❷
except AttributeError: ❸
```

```
    pass ❹
field_names = tuple(field_names) ❺
```

❶ 假设是单个字符串（EAFP 风格，即“取得原谅比获得许可容易”）。

❷ 把逗号替换成空格，然后拆分成名称列表。

❸ 抱歉，`field_names` 看起来不像是字符串……没有 `.replace` 方法，或者返回值不能使用 `.split` 方法拆分。

❹ 假设已经是由名称组成的可迭代对象了。

❺ 为了确保的确是可迭代对象，也为了保存一份副本，使用所得值创建一个元组。

在那篇短文的最后，Alex 多次强调，要抑制住创建抽象基类的冲动。滥用抽象基类会造成灾难性后果，表明语言太注重表面形式，这对以实用和务实著称的 Python 可不是好事。在审阅本书的过程中，Alex 写道：

> 抽象基类是用于封装框架引入的一般性概念和抽象的，例如“一个序列”和“一个确切的数”。（读者）基本上不需要自己编写新的抽象基类，只要正确使用现有的抽象基类，就能获得 99.9% 的好处，而不用冒着设计不当导致的巨大风险。

下面通过实例讲解白鹅类型。

11.5　定义抽象基类的子类

我们将遵循 Martelli 的建议，先利用现有的抽象基类（collections.MutableSequence），然后再斗胆自己定义。在示例 11-8 中，我们明确把 FrenchDeck2 声明为 collections.MutableSequence 的子类。

示例 **11-8** frenchdeck2.py：FrenchDeck2，collections.MutableSequence 的子类

```
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck2(collections.MutableSequence):
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                                        for rank in self.ranks]

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, position):
        return self._cards[position]

    def __setitem__(self, position, value):  # ❶
        self._cards[position] = value

    def __delitem__(self, position):  # ❷
        del self._cards[position]

    def insert(self, position, value):  # ❸
        self._cards.insert(position, value)
```

❶ 为了支持洗牌，只需实现 `__setitem__` 方法。

❷ 但是继承 `MutableSequence` 的类必须实现 `__delitem__` 方法，这是 `MutableSequence` 类的一个抽象方法。

❸ 此外，还要实现 `insert` 方法，这是 `MutableSequence` 类的第三个抽象方法。

导入时（加载并编译 frenchdeck2.py 模块时），Python 不会检查抽象方法的实现，在运行时实例化 `FrenchDeck2` 类时才会真正检查。因此，如果没有正确实现某个抽象方法，Python 会抛出 `TypeError` 异常，并把错误消息设为`"Can't instantiate abstract class FrenchDeck2` with abstract methods `__delitem__, insert"`。正是这个原因，即便 `FrenchDeck2` 类不需要 `__delitem__` 和 `insert` 提供的行为，也要实现，因为 `MutableSequence` 抽象基类需要它们。

如图 11-2 所示，`Sequence` 和 `MutableSequence` 抽象基类的方法不全是抽象的。

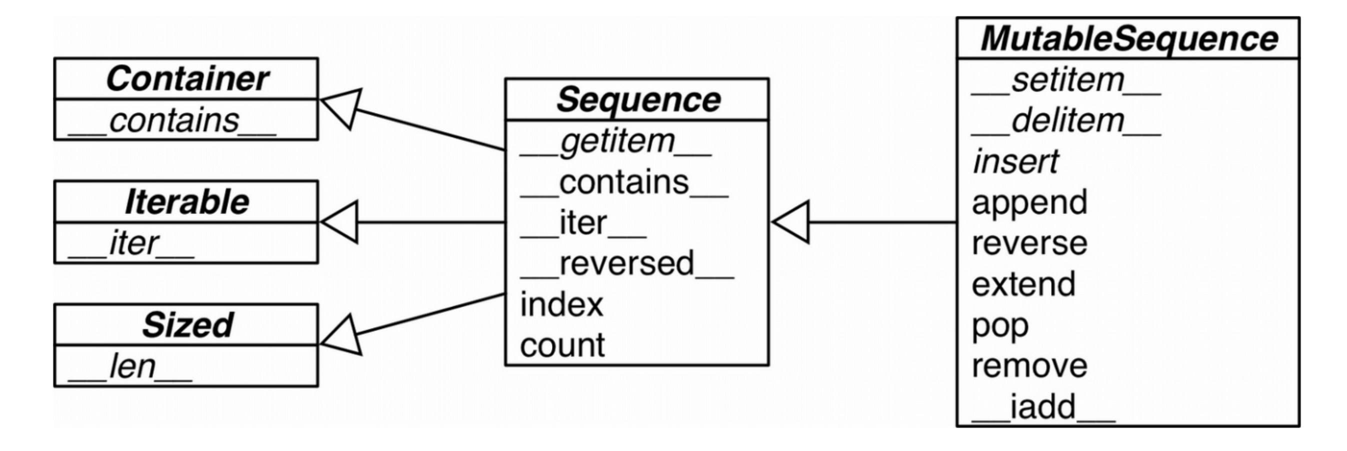


图 **11-2**：**`MutableSequence`** 抽象基类和 **`collections.abc`** 中它的超类的 **UML** 类图（箭头由子类指向祖先；以斜体显示的名称是抽象类和抽象方法）

`FrenchDeck2` 从 `Sequence` 继承了几个拿来即用的具体方法：`__contains__`、`__iter__`、`__reversed__`、`index` 和 `count`。`FrenchDeck2` 从 `MutableSequence` 继承了 `append`、`extend`、`pop`、`remove` 和`__iadd__`。

在 `collections.abc` 中，每个抽象基类的具体方法都是作为类的公开

接口实现的，因此不用知道实例的内部接口。

> 要想实现子类，我们可以覆盖从抽象基类中继承的方法，以更高效的方式重新实现。例如，`__contains__` 方法会全面扫描序列，可是，如果你定义的序列按顺序保存元素，那就可以重新定义 `__contains__` 方法，使用 `bisect` 函数做二分查找（参见 2.8 节），从而提升搜索速度。

为了充分使用抽象基类，我们要知道有哪些抽象基类可用。接下来介绍集合抽象基类。

11.6　标准库中的抽象基类

从 Python 2.6 开始，标准库提供了抽象基类。大多数抽象基类在 `collections.abc` 模块中定义，不过其他地方也有。例如，`numbers` 和 `io` 包中有一些抽象基类。但是，`collections.abc` 中的抽象基类最常用。我们来看看这个模块中有哪些抽象基类。

11.6.1　collections.abc模块中的抽象基类

标准库中有两个名为 `abc` 的模块，这里说的是 `collections.abc`。为了减少加载时间，Python 3.4 在 `collections` 包之外实现这个模块（在 Lib/_collections_abc.py 中，https://hg.python.org/cpython/file/3.4/Lib/_collections_abc.py），因此要与 `collections` 分开导入。另一个 `abc` 模块就是 `abc`（即 Lib/abc.py，https://hg.python.org/cpython/file/3.4/Lib/abc.py），这里定义的是 `abc.ABC` 类。每个抽象基类都依赖这个类，但是不用导入它，除非定义新抽象基类。

Python 3.4 在 `collections.abc` 模块中定义了 16 个抽象基类，简要的 UML 类图（没有属性名称）如图 11-3 所示。`collections.abc` 的官方文档中有个不错的表格（https://docs.python.org/3/library/collections.abc.html#collections-abstract-base-classes），对各个抽象基类做了总结，说明了相互之间的关系，以及各个基类提供的抽象方法和具体方法（称为“混入方法”）。图 11-3 中有很多多重继承。我们将在第 12 章着重说明多重继承，讨论抽象基类时通常不用考虑多重继承。[6]

[6]Java 认为多重继承有危害，因此没有提供支持，但是提供了接口：Java 的接口可以扩展多个接口，而且 Java 的类可以实现多个接口。

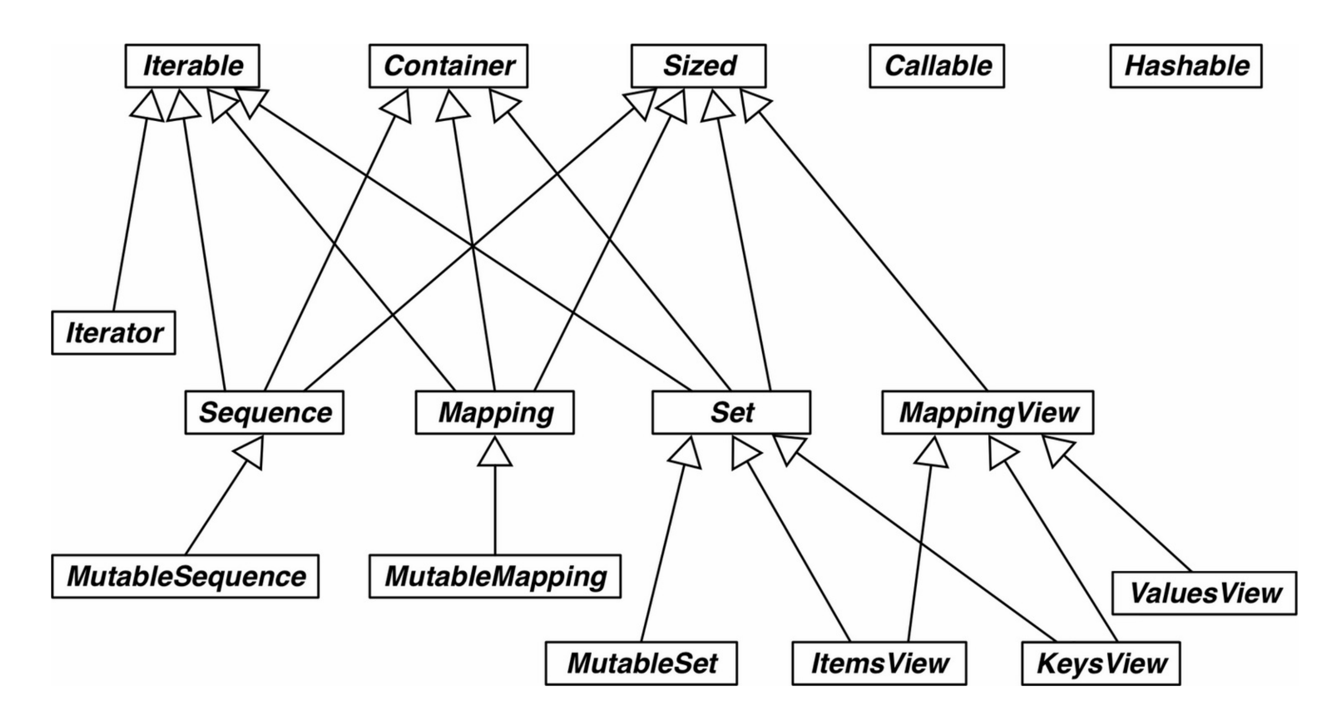


图 **11-3**：**`collections.abc`** 模块中各个抽象基类的 **UML** 类图

下面详述图 11-3 中那一群基类。

`Iterable`、`Container` 和 `Sized`

　　各个集合应该继承这三个抽象基类，或者至少实现兼容的协议。`Iterable` 通过 `__iter__` 方法支持迭代，`Container` 通过 `__contains__` 方法支持 `in` 运算符，`Sized` 通过 `__len__` 方法支持 `len()` 函数。

`Sequence`、`Mapping` 和 `Set`

　　这三个是主要的不可变集合类型，而且各自都有可变的子类。`MutableSequence` 的详细类图见图 11-2；`MutableMapping` 和 `MutableSet` 的类图在第 3 章中（见图 3-1 和图 3-2）。

`MappingView`

　　在 Python 3 中，映射方法 `.items()`、`.keys()` 和 `.values()` 返回的对象分别是 `ItemsView`、`KeysView` 和 `ValuesView` 的实例。前两个类还从 `Set` 类继承了丰富的接口，包含 3.8.3 节所述的全部运算符。

`Callable` 和 `Hashable`

这两个抽象基类与集合没有太大的关系，只不过因为 `collections.abc` 是标准库中定义抽象基类的第一个模块，而它们又太重要了，因此才把它们放到 `collections.abc` 模块中。我从未见过 `Callable` 或 `Hashable` 的子类。这两个抽象基类的主要作用是为内置函数 `isinstance` 提供支持，以一种安全的方式判断对象能不能调用或散列。[7]

> [7]若想检查是否能调用，可以使用内置的 `callable()` 函数；但是没有类似的 `hashable()` 函数，因此测试对象是否可散列，最好使用 `isinstance(my_obj, Hashable)`。

`Iterator`

注意它是 `Iterable` 的子类。我们将在第 14 章详细讨论。

继 `collections.abc` 之后，标准库中最有用的抽象基类包是 `numbers`。下面就来介绍。

11.6.2　抽象基类的数字塔

`numbers` 包（https://docs.python.org/3/library/numbers.html）定义的是“数字塔”（即各个抽象基类的层次结构是线性的），其中 `Number` 是位于最顶端的超类，随后是 `Complex` 子类，依次往下，最底端是 `Integral` 类：

- `Number`
- `Complex`
- `Real`
- `Rational`
- `Integral`

因此，如果想检查一个数是不是整数，可以使用 `isinstance(x,`

numbers.Integral)，这样代码就能接受 int、bool（int 的子类），或者外部库使用 numbers 抽象基类注册的其他类型。为了满足检查的需要，你或者你的 API 的用户始终可以把兼容的类型注册为 numbers.Integral 的虚拟子类。

与之类似，如果一个值可能是浮点数类型，可以使用 isinstance(x, numbers.Real) 检查。这样代码就能接受 bool、int、float、fractions.Fraction，或者外部库（如 NumPy，它做了相应的注册）提供的非复数类型。

> decimal.Decimal 没有注册为 numbers.Real 的虚拟子类，这有点奇怪。没注册的原因是，如果你的程序需要 Decimal 的精度，要防止与其他低精度数字类型混淆，尤其是浮点数。

了解一些现有的抽象基类之后，我们将从零开始实现一个抽象基类，然后实际使用，以此实践白鹅类型。这么做的目的不是鼓励每个人都立即开始定义抽象基类，而是教你怎么阅读标准库和其他包中的抽象基类源码。

11.7　定义并使用一个抽象基类

为了证明有必要定义抽象基类，我们要在框架中找到使用它的场景。想象一下这个场景：你要在网站或移动应用中显示随机广告，但是在整个广告清单轮转一遍之前，不重复显示广告。假设我们在构建一个广告管理框架，名为 ADAM。它的职责之一是，支持用户提供随机挑选的无重复类。[8] 为了让 ADAM 的用户明确理解“随机挑选的无重复”组件是什么意思，我们将定义一个抽象基类。

> [8]客户可能要审查随机发生器，或者代理想作弊……谁知道呢！

受到“栈”和“队列”（以物体的排放方式说明抽象接口）启发，我将使用现实世界中的物品命名这个抽象基类：宾果机和彩票机是随机从有限的集合中挑选物品的机器，选出的物品没有重复，直到选完为止。

我们把这个抽象基类命名为 `Tombola`，这是宾果机和打乱数字的滚动容器的意大利名。[9]

> [9]牛津英语词典对 tombola 的定义是“像对号游戏（lotto）那样的彩票（lottery）”。

`Tombola` 抽象基类有四个方法，其中两个是抽象方法。

- `.load(...)`：把元素放入容器。
- `.pick()`：从容器中随机拿出一个元素，返回选中的元素。

另外两个是具体方法。

- `.loaded()`：如果容器中至少有一个元素，返回 `True`。
- `.inspect()`：返回一个有序元组，由容器中的现有元素构成，不会修改容器的内容（内部的顺序不保留）。

图 11-4 展示了 `Tombola` 抽象基类和三个具体实现。

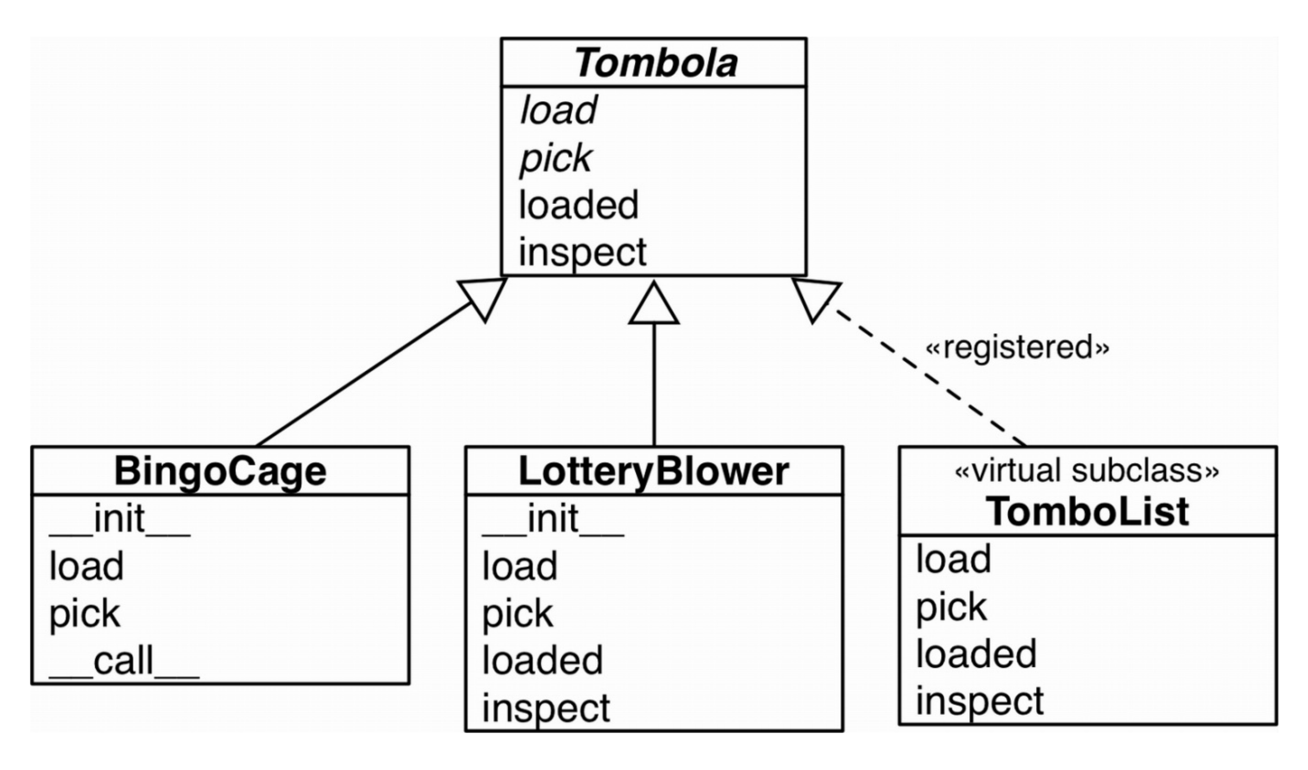


图 **11-4**：一个抽象基类和三个子类的 **UML** 类图。根据 **UML** 的约定，**`Tombola`** 抽象基类和它的抽象方法使用斜体。虚线箭头用于表示接口实现，这里它表示 **`TomboList`** 是 **`Tombola`** 的虚拟子类，因为 **`TomboList`** 是注册的，本章后面会说明这一点 [10]

[10]«registered» 和«virtual subclass» 不是标准的 UML 词汇。我们使用二者表示 Python 类之间的关系。

`Tombola` 抽象基类的定义如示例 11-9 所示。

示例 **11-9**　tombola.py：`Tombola` 是抽象基类，有两个抽象方法和两个具体方法

```
import abc

class Tombola(abc.ABC):  ❶

    @abc.abstractmethod
    def load(self, iterable):  ❷
        """从可迭代对象中添加元素。"""

    @abc.abstractmethod
    def pick(self):  ❸
```

```
        """随机删除元素，然后将其返回。

        如果实例为空，这个方法应该抛出`LookupError`。
        """

    def loaded(self):  ❹
        """如果至少有一个元素，返回`True`，否则返回`False`。"""
        return bool(self.inspect())  ❺


    def inspect(self):
        """返回一个有序元组，由当前元素构成。"""
        items = []
        while True:  ❻
            try:
                items.append(self.pick())
            except LookupError:
                break
        self.load(items)  ❼
        return tuple(sorted(items))
```

❶ 自己定义的抽象基类要继承 `abc.ABC`。

❷ 抽象方法使用 `@abstractmethod` 装饰器标记，而且定义体中通常只有文档字符串。[11]

> [11]在抽象基类出现之前，抽象方法使用 `raise NotImplementedError` 语句表明由子类负责实现。

❸ 根据文档字符串，如果没有元素可选，应该抛出 `LookupError`。

❹ 抽象基类可以包含具体方法。

❺ 抽象基类中的具体方法只能依赖抽象基类定义的接口（即只能使用抽象基类中的其他具体方法、抽象方法或特性）。

❻ 我们不知道具体子类如何存储元素，不过为了得到 `inspect` 的结果，我们可以不断调用 `.pick()` 方法，把 `Tombola` 清空……

❼ ……然后再使用 `.load(...)` 把所有元素放回去。

其实，抽象方法可以有实现代码。即便实现了，子类也必须覆盖抽象方法，但是在子类中可以使用 `super()` 函数调用抽象方法，为它添加功能，而不是从头开始实现。`@abstractmethod` 装饰器的用法参见 `abc` 模块的文档（https://docs.python.org/3/library/abc.html）。

示例 11-9 中的 `.inspect()` 方法实现的方式有些笨拙，不过却表明，有了 `.pick()` 和 `.load(…)` 方法，若想查看 `Tombola` 中的内容，可以先把所有元素挑出，然后再放回去。这个示例的目的是强调抽象基类可以提供具体方法，只要依赖接口中的其他方法就行。`Tombola` 的具体子类知晓内部数据结构，可以覆盖 `.inspect()` 方法，使用更聪明的方式实现，但这不是强制要求。

示例 11-9 中的 `.loaded()` 方法没有那么笨拙，但是耗时：调用 `.inspect()` 方法构建有序元组的目的仅仅是在其上调用 `bool()` 函数。这样做是可以的，但是具体子类可以做得更好，后文见分晓。

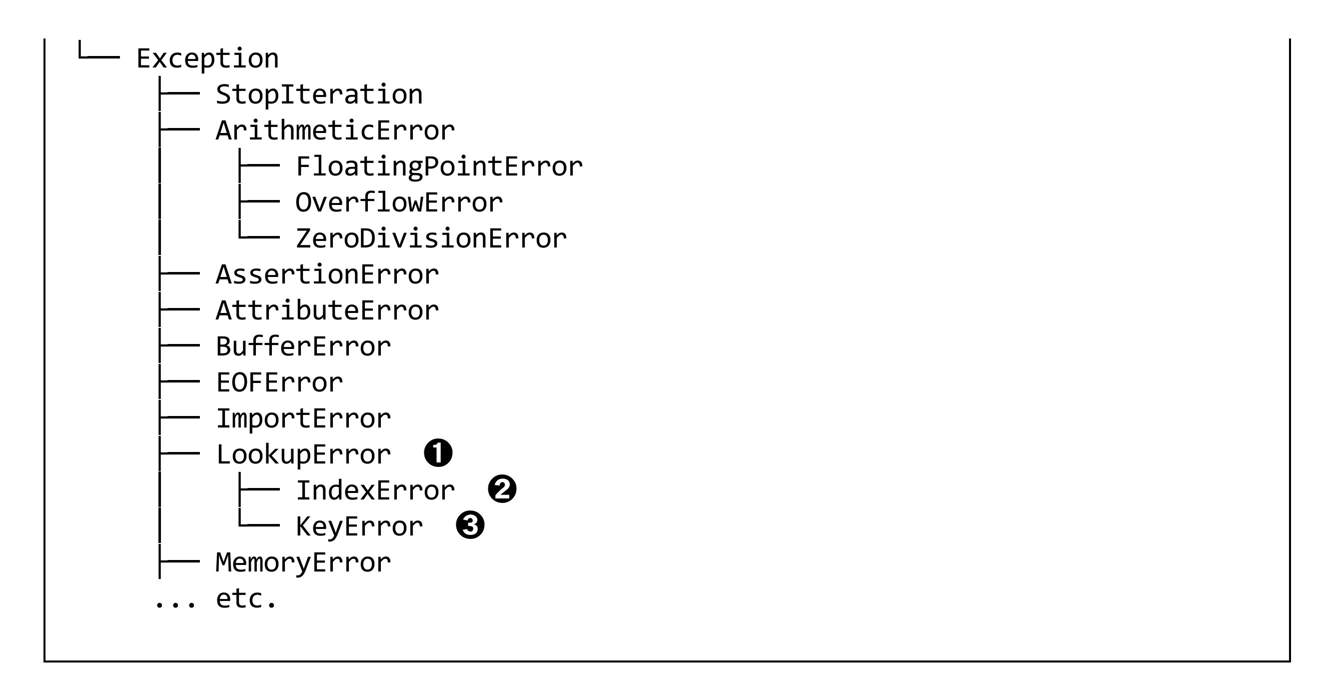
注意，实现 `.inspect()` 方法采用的迂回方式要求捕获 `self.pick()` 抛出的 `LookupError`。`self.pick()` 抛出 `LookupError` 这一事实也是接口的一部分，但是在 Python 中没办法声明，只能在文档中说明（参见示例 11-9 中抽象方法 `pick` 的文档字符串）。

我选择使用 `LookupError` 异常的原因是，在 Python 的异常层次关系中，它与 `IndexError` 和 `KeyError` 有关，这两个是具体实现 `Tombola` 所用的数据结构最有可能抛出的异常。据此，实现代码可能会抛出 `LookupError`、`IndexError` 或 `KeyError` 异常。异常的部分层次结构如示例 11-10 所示（完整的层次结构参见 Python 标准库文档中的“5.4. Exception hierarchy”一节。[12]）

[12] 见 https://docs.python.org/dev/library/exceptions.html#exception-hierarchy。——编者注

示例 **11-10**　异常类的部分层次结构

```
BaseException
 ├── SystemExit
 ├── KeyboardInterrupt
 ├── GeneratorExit
```



```
└── Exception
     ├── StopIteration
     ├── ArithmeticError
     │    ├── FloatingPointError
     │    ├── OverflowError
     │    └── ZeroDivisionError
     ├── AssertionError
     ├── AttributeError
     ├── BufferError
     ├── EOFError
     ├── ImportError
     ├── LookupError  ❶
     │    ├── IndexError  ❷
     │    └── KeyError  ❸
     ├── MemoryError
    ... etc.
```

❶ 我们在 `Tombola.inspect` 方法中处理的是 `LookupError` 异常。

❷ `IndexError` 是 `LookupError` 的子类，尝试从序列中获取索引超过最后位置的元素时抛出。

❸ 使用不存在的键从映射中获取元素时，抛出 `KeyError` 异常。

我们自己定义的 `Tombola` 抽象基类完成了。为了一睹抽象基类对接口所做的检查，下面我们尝试使用一个有缺陷的实现来糊弄 `Tombola`，如示例 11-11 所示。

示例 **11-11**　不符合 `Tombola` 要求的子类无法蒙混过关

```
>>> from tombola import Tombola
>>> class Fake(Tombola):  # ❶
...     def pick(self):
...         return 13
...
>>> Fake  # ❷
<class '__main__.Fake'>
>>> f = Fake()  # ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Fake with abstract methods load
```

❶ 把 Fake 声明为 Tombola 的子类。

❷ 创建了 Fake 类，目前没有错误。

❸ 尝试实例化 Fake 时抛出了 TypeError。错误消息十分明确：Python 认为 Fake 是抽象类，因为它没有实现 load 方法，这是 Tombola 抽象基类声明的抽象方法之一。

我们的第一个抽象基类定义好了，而且还用它实际验证了一个类。稍后我们将定义 Tombola 抽象基类的子类，在此之前必须说明抽象基类的一些编程规则。

11.7.1　抽象基类句法详解

声明抽象基类最简单的方式是继承 abc.ABC 或其他抽象基类。

然而，abc.ABC 是 Python 3.4 新增的类，因此如果你使用的是旧版 Python，那么无法继承现有的抽象基类。此时，必须在 class 语句中使用 metaclass= 关键字，把值设为 abc.ABCMeta（不是 abc.ABC）。在示例 11-9 中，可以写成：

```
class Tombola(metaclass=abc.ABCMeta):
    # ...
```

metaclass= 关键字参数是 Python 3 引入的。在 Python 2 中必须使用 __metaclass__ 类属性：

```
class Tombola(object):  # 这是Python 2！！！
    __metaclass__ = abc.ABCMeta
    # ...
```

元类将在第 21 章讲解。现在，我们暂且把元类理解为一种特殊的类，同样也把抽象基类理解为一种特殊的类。例如，“常规的”类不会检查子类，因此这是抽象基类的特殊行为。

除了 @abstractmethod 之外，abc 模块还定义了

@abstractclassmethod、@abstractstaticmethod 和 @abstractproperty 三个装饰器。然而，后三个装饰器从 Python 3.3 起废弃了，因为装饰器可以在 @abstractmethod 上堆叠，那三个就显得多余了。例如，声明抽象类方法的推荐方式是：

```
class MyABC(abc.ABC):
    @classmethod
    @abc.abstractmethod
    def an_abstract_classmethod(cls, ...):
        pass
```



在函数上堆叠装饰器的顺序通常很重要，@abstractmethod 的文档就特别指出：

> 与其他方法描述符一起使用时，abstractmethod() 应该放在最里层，……[13]

也就是说，在 @abstractmethod 和 def 语句之间不能有其他装饰器。

[13]出自 abc 模块文档中的 @abc.abstractmethod 词条（https://docs.python.org/dev/library/abc.html#abc.abstractmethod）。

说明抽象基类的句法之后，我们要通过实现几个功能完善的具体子代来使用 Tombola。

11.7.2　定义Tombola抽象基类的子类

定义好 Tombola 抽象基类之后，我们要开发两个具体子类，满足 Tombola 规定的接口。这两个子类的类图如图 11-4 所示，图中还有将在下一节讨论的虚拟子类。

示例 11-12 中的 BingoCage 类是在示例 5-8 的基础上修改的，使用了更好的随机发生器。 BingoCage 实现了所需的抽象方法 load 和 pick，

从 Tombola 中继承了 loaded 方法，覆盖了 inspect 方法，还增加了 __call__ 方法。

示例 **11-12** bingo.py：BingoCage 是 Tombola 的具体子类

```
import random

from tombola import Tombola


class BingoCage(Tombola):  ❶

    def __init__(self, items):
        self._randomizer = random.SystemRandom()  ❷
        self._items = []
        self.load(items)  ❸

    def load(self, items):
        self._items.extend(items)
        self._randomizer.shuffle(self._items)  ❹

    def pick(self):  ❺
        try:
            return self._items.pop()
        except IndexError:
            raise LookupError('pick from empty BingoCage')

    def __call__(self):  ❻
        self.pick()
```

❶ 明确指定 BingoCage 类扩展 Tombola 类。

❷ 假设我们将在线上游戏中使用这个。random.SystemRandom 使用 os.urandom(...) 函数实现 random API。根据 os 模块的文档（http://docs.python.org/3/library/os.html#os.urandom），os.urandom(... 函数生成“适合用于加密”的随机字节序列。

❸ 委托 .load(...) 方法实现初始加载。

❹ 没有使用 random.shuffle() 函数，而是使用 SystemRandom 实例的 .shuffle() 方法。

❺ `pick` 方法的实现方式与示例 5-8 一样。

❻ `__call__` 也跟示例 5-8 中的一样。它没必要满足 `Tombola` 接口，添加额外的方法没有问题。

`BingoCage` 从 `Tombola` 中继承了耗时的 `loaded` 方法和笨拙的 `inspect` 方法。这两个方法都可以覆盖，变成示例 11-13 中速度更快的一行代码。这里想表达的观点是：我们可以偷懒，直接从抽象基类中继承不是那么理想的具体方法。从 `Tombola` 中继承的方法没有 `BingoCage` 自己定义的那么快，不过只要 `Tombola` 的子类正确实现 `pick` 和 `load` 方法，就能提供正确的结果。

示例 11-13 是 `Tombola` 接口的另一种实现，虽然与之前不同，但完全有效。`LotteryBlower` 打乱“数字球”后没有取出最后一个，而是取出一个随机位置上的球。

示例 **11-13**　lotto.py：`LotteryBlower` 是 `Tombola` 的具体子类，覆盖了继承的 `inspect` 和 `loaded` 方法

```
import random

from tombola import Tombola


class LotteryBlower(Tombola):

    def __init__(self, iterable):
        self._balls = list(iterable)  ❶

    def load(self, iterable):
        self._balls.extend(iterable)

    def pick(self):
        try:
            position = random.randrange(len(self._balls))  ❷
        except ValueError:
            raise LookupError('pick from empty LotteryBlower')
        return self._balls.pop(position) ❸

    def loaded(self):  ❹
        return bool(self._balls)
```

```
    def inspect(self):  ❺
        return tuple(sorted(self._balls))
```

❶ 初始化方法接受任何可迭代对象：把参数构建成列表。

❷ 如果范围为空，`random.randrange(...)` 函数抛出 `ValueError`，为了兼容 `Tombola`，我们捕获它，抛出 `LookupError`。

❸ 否则，从 `self._balls` 中取出随机选中的元素。

❹ 覆盖 `loaded` 方法，避免调用 `inspect` 方法（示例 11-9 中的 `Tombola.loaded` 方法是这么做的）。我们可以直接处理 `self._balls` 而不必构建整个有序元组，从而提升速度。

❺ 使用一行代码覆盖 `inspect` 方法。

示例 11-13 中有个习惯做法值得指出：在 `__init__` 方法中，`self._balls` 保存的是 `list(iterable)`，而不是 `iterable` 的引用（即没有直接把 `iterable` 赋值给 `self._balls`）。前面说过，[14] 这样做使得 `LotteryBlower` 更灵活，因为 `iterable` 参数可以是任何可迭代的类型。把元素存入列表中还确保能取出元素。就算 `iterable` 参数始终传入列表，`list(iterable)` 会创建参数的副本，这依然是好的做法，因为我们要从中删除元素，而客户可能不希望自己提供的列表被修改。[15]

[14]我在 Martelli 写的"水禽和抽象基类"短文之后以此为例说明鸭子类型。

[15].4.2 节专门讨论了这种防止混淆别名的问题。

接下来要讲白鹅类型的重要动态特性了：使用 `register` 方法声明虚拟子类。

11.7.3　`Tombola`的虚拟子类

白鹅类型的一个基本特性（也是值得用水禽来命名的原因）：即便不继承，也有办法把一个类注册为抽象基类的虚拟子类。这样做时，我们保

证注册的类忠实地实现了抽象基类定义的接口，而 Python 会相信我们，从而不做检查。如果我们说谎了，那么常规的运行时异常会把我们捕获。

注册虚拟子类的方式是在抽象基类上调用 `register` 方法。这么做之后，注册的类会变成抽象基类的虚拟子类，而且 `issubclass` 和 `isinstance` 等函数都能识别，但是注册的类不会从抽象基类中继承任何方法或属性。

> 虚拟子类不会继承注册的抽象基类，而且任何时候都不会检查它是否符合抽象基类的接口，即便在实例化时也不会检查。为了避免运行时错误，虚拟子类要实现所需的全部方法。

`register` 方法通常作为普通的函数调用（参见 11.9 节），不过也可以作为装饰器使用。在示例 11-14 中，我们使用装饰器句法实现了 `TomboList` 类，这是 `Tombola` 的一个虚拟子类，如图 11-5 所示。

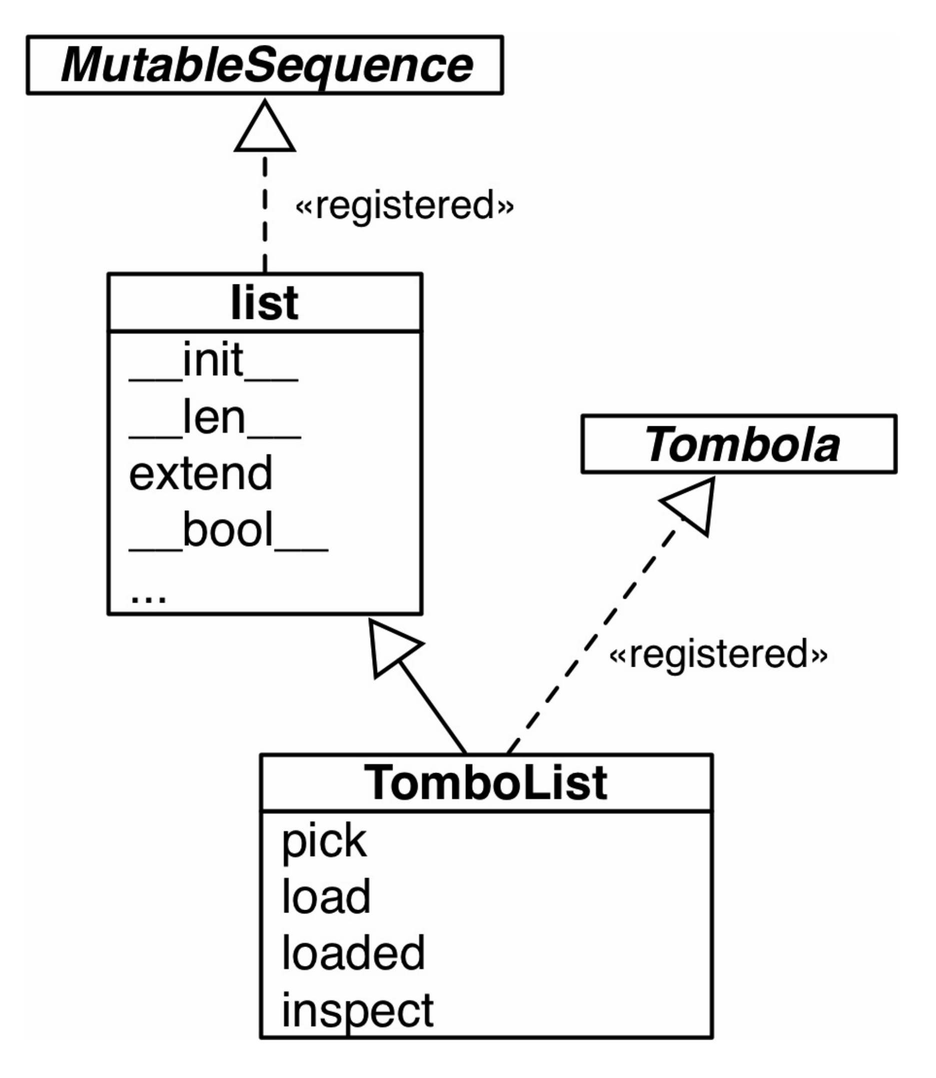


图 **11-5**：**`TomboList`** 的 **UML** 类图，它是 **`list`** 的真实子类和 **`Tombola`** 的虚拟子类

`TomboList` 能像它宣称的那样使用，doctest 能证明这一点，详情参见 11.8 节。

示例 **11-14**　tombolist.py：`TomboList` 是 `Tombola` 的虚拟子类

```
from random import randrange

from tombola import Tombola

@Tombola.register  # ❶
class TomboList(list):  # ❷

    def pick(self):
```

```
        if self:  # ❸
            position = randrange(len(self))
            return self.pop(position)  # ❹
        else:
            raise LookupError('pop from empty TomboList')

    load = list.extend  # ❺

    def loaded(self):
        return bool(self)  # ❻

    def inspect(self):
        return tuple(sorted(self))

# Tombola.register(TomboList)  # ❼
```

❶ 把 `Tombolist` 注册为 `Tombola` 的虚拟子类。

❷ `Tombolist` 扩展 `list`。

❸ `Tombolist` 从 `list` 中继承 `__bool__` 方法，列表不为空时返回 `True`。

❹ `pick` 调用继承自 `list` 的 `self.pop` 方法，传入一个随机的元素索引。

❺ `Tombolist.load` 与 `list.extend` 一样。

❻ `loaded` 方法委托 `bool` 函数。[16]

> [16] `loaded` 方法不能采用 `load` 方法的那种方式，因为 `list` 类型没有实现 `loaded` 方法所需的 `__bool__` 方法。而内置的 `bool` 函数不需要 `__bool__` 方法，因为它还可以使用 `__len__` 方法。参见 Python 文档中“Built-in Types”一章中的“4.1. Truth Value Testing”（https://docs.python.org/3/library/stdtypes.html#truth）。

❼ 如果是 Python 3.3 或之前的版本，不能把 `.register` 当作类装饰器使用，必须使用标准的调用句法。

注册之后，可以使用 `issubclass` 和 `isinstance` 函数判断 `TomboList` 是不是 `Tombola` 的子类：

```
>>> from tombola import Tombola
>>> from tombolist import TomboList
>>> issubclass(TomboList, Tombola)
True
>>> t = TomboList(range(100))
>>> isinstance(t, Tombola)
True
```

然而，类的继承关系在一个特殊的类属性中指定—— `__mro__`，即方法解析顺序（Method Resolution Order）。这个属性的作用很简单，按顺序列出类及其超类，Python 会按照这个顺序搜索方法。[17] 查看 `TomboList` 类的 `__mro__` 属性，你会发现它只列出了“真实的”超类，即 `list` 和 `object`：

> [17]12.2 节会专门讲解 `__mro__` 类属性，现在知道这个简单的解释就行了。

```
>>> TomboList.__mro__
(<class 'tombolist.TomboList'>, <class 'list'>, <class 'object'>)
```

`Tombolist.__mro__` 中没有 `Tombola`，因此 `Tombolist` 没有从 `Tombola` 中继承任何方法。

我编写了几个类，实现了相同的接口，现在我需要一种编写 doctest 的方式来涵盖不同的实现。下一节说明如何利用常规类和抽象基类的 API 编写 doctest。

11.8　Tombola子类的测试方法

我编写的 Tombola 示例测试脚本用到两个类属性，用它们内省类的继承关系。

`__subclasses__()`

　　这个方法返回类的直接子类列表，不含虚拟子类。

`_abc_registry`

　　只有抽象基类有这个数据属性，其值是一个 WeakSet 对象，即抽象类注册的虚拟子类的弱引用。

为了测试 Tombola 的所有子类，我编写的脚本迭代 `Tombola.__subclasses__()` 和 `Tombola._abc_registry` 得到的列表，然后把各个类赋值给在 doctest 中使用的 ConcreteTombola。

这个测试脚本成功运行时输出的结果如下：

```
$ python3 tombola_runner.py
BingoCage        24 tests,  0 failed - OK
LotteryBlower    24 tests,  0 failed - OK
TumblingDrum     24 tests,  0 failed - OK
TomboList        24 tests,  0 failed - OK
```

测试脚本的代码在示例 11-15 中，doctest 在示例 11-16 中。

示例 **11-15**　tombola_runner.py：Tombola 子类的测试运行程序

```
import doctest

from tombola import Tombola

# 要测试的模块
import bingo, lotto, tombolist, drum  ❶
```

```
TEST_FILE = 'tombola_tests.rst'
TEST_MSG = '{0:16} {1.attempted:2} tests, {1.failed:2} failed - {2}'


def main(argv):
    verbose = '-v' in argv
    real_subclasses = Tombola.__subclasses__()  ❷
    virtual_subclasses = list(Tombola._abc_registry)  ❸

    for cls in real_subclasses + virtual_subclasses:  ❹
        test(cls, verbose)


def test(cls, verbose=False):

    res = doctest.testfile(
            TEST_FILE,
            globs={'ConcreteTombola': cls},  ❺
            verbose=verbose,
            optionflags=doctest.REPORT_ONLY_FIRST_FAILURE)
    tag = 'FAIL' if res.failed else 'OK'
    print(TEST_MSG.format(cls.__name__, res, tag))  ❻

if __name__ == '__main__':
    import sys
    main(sys.argv)
```

❶ 导入包含 `Tombola` 真实子类和虚拟子类的模块，用于测试。

❷ `__subclasses__()` 返回的列表是内存中存在的直接子代。即便源码中用不到想测试的模块，也要将其导入，因为要把那些类载入内存。

❸ 把 `_abc_registry`（`WeakSet` 对象）转换成列表，这样方能与 `__subclasses__()` 的结果拼接起来。

❹ 迭代找到的各个子类，分别传给 `test` 函数。

❺ 把 `cls` 参数（要测试的类）绑定到全局命名空间里的 `ConcreteTombola` 名称上，供 doctest 使用。

❻ 输出测试结果，包含类的名称、尝试运行的测试数量、失败的测试数量，以及 `'OK'` 或 `'FAIL'` 标记。

doctest 文件如示例 11-16 所示。

示例 **11-16**　tombola_tests.rst：`Tombola` 子类的 doctest

```
==============
Tombola tests
==============

Every concrete subclass of Tombola should pass these tests.


Create and load instance from iterable::

    >>> balls = list(range(3))
    >>> globe = ConcreteTombola(balls)
    >>> globe.loaded()
    True
    >>> globe.inspect()
    (0, 1, 2)


Pick and collect balls::

    >>> picks = []
    >>> picks.append(globe.pick())
    >>> picks.append(globe.pick())
    >>> picks.append(globe.pick())


Check state and results::

    >>> globe.loaded()
    False
    >>> sorted(picks) == balls
    True


Reload::

    >>> globe.load(balls)
    >>> globe.loaded()
    True
    >>> picks = [globe.pick() for i in balls]
    >>> globe.loaded()
    False
```

```
Check that `LookupError` (or a subclass) is the exception
thrown when the device is empty::

    >>> globe = ConcreteTombola([])
    >>> try:
    ... globe.pick()
    ... except LookupError as exc:
    ... print('OK')
    OK


Load and pick 100 balls to verify that they all come out::

    >>> balls = list(range(100))
    >>> globe = ConcreteTombola(balls)
    >>> picks = []
    >>> while globe.inspect():
    ... picks.append(globe.pick())
    >>> len(picks) == len(balls)
    True
    >>> set(picks) == set(balls)
    True


Check that the order has changed and is not simply reversed::
    >>> picks != balls
    True
    >>> picks[::-1] != balls
    True

Note: the previous 2 tests have a *very* small chance of failing
even if the implementation is OK. The probability of the 100
balls coming out, by chance, in the order they were inspect is
1/100!, or approximately 1.07e-158. It's much easier to win the
Lotto or to become a billionaire working as a programmer.

THE END
```

我们对 `Tombola` 抽象基类的分析到此结束。下一节说明 Python 如何使用抽象基类的 `register` 函数。

11.9　Python使用register的方式

示例 11-14 把 Tombola.register 当作类装饰器使用。在 Python 3.3 之前的版本中不能这样使用 register，必须在定义类之后像普通函数那样调用，如示例 11-14 中最后那行注释所述。

虽然现在可以把 register 当作装饰器使用了，但更常见的做法还是把它当作函数使用，用于注册其他地方定义的类。例如，在 collections.abc 模块的源码中（https://hg.python.org/cpython/file/3.4/Lib/_collections_abc.py），是这样把内置类型 tuple、str、range 和 memoryview 注册为 Sequence 的虚拟子类的：

```
Sequence.register(tuple)
Sequence.register(str)
Sequence.register(range)
Sequence.register(memoryview)
```

其他几个内置类型在 _collections_abc.py 文件（https://hg.python.org/cpython/file/3.4/Lib/_collections_abc.py）中注册为抽象基类的虚拟子类。这些类型在导入模块时注册，这样做是可以的，因为必须导入才能使用抽象基类：能访问 MutableMapping 才能编写 isinstance(my_dict, MutableMapping)。

结束本章之前，还要解释一下 Alex Martelli 在“水禽和抽象基类”中施展的魔法。

11.10　鹅的行为有可能像鸭子

Alex 在他写的“水禽和抽象基类”一文中指出，即便不注册，抽象基类也能把一个类识别为虚拟子类。下面是他举的例子，我添加了一些代码，使用 `issubclass` 做测试：

```
>>> class Struggle:
...     def __len__(self): return 23
...
>>> from collections import abc
>>> isinstance(Struggle(), abc.Sized)
True
>>> issubclass(Struggle, abc.Sized)
True
```

经 `issubclass` 函数确认（`isinstance` 函数也会得出相同的结论），`Struggle` 是 `abc.Sized` 的子类，这是因为 `abc.Sized` 实现了一个特殊的类方法，名为 `__subclasshook__`。参见示例 11-17。

示例 **11-17**　`Sized` 类的源码，摘自 Lib/_collections_abc.py（Python 3.4，https://hg.python.org/cpython/file/3.4/Lib/_collections_abc.py#l127）

```
class Sized(metaclass=ABCMeta):

    __slots__ = ()

    @abstractmethod
    def __len__(self):
        return 0

    @classmethod
    def __subclasshook__(cls, C):
        if cls is Sized:
            if any("__len__" in B.__dict__ for B in C.__mro__):  # ❶
                return True  # ❷
        return NotImplemented  # ❸
```

❶ 对 `C.__mro__` （即 `C` 及其超类）中所列的类来说，如果类的

__dict__ 属性中有名为 __len__ 的属性……

❷ ……返回 True，表明 C 是 Sized 的虚拟子类。

❸ 否则，返回 NotImplemented，让子类检查。

如果你对子类检查的细节感兴趣，可以阅读 Lib/abc.py 文件中 ABCMeta.__subclasscheck__ 方法的源码（https://hg.python.org/cpython/file/3.4/Lib/abc.py#l194）。提醒：源码中有很多 if 语句和两个递归调用。

__subclasshook__ 在白鹅类型中添加了一些鸭子类型的踪迹。我们可以使用抽象基类定义正式接口，可以始终使用 isinstance 检查，也可以完全使用不相关的类，只要实现特定的方法即可（或者做些事情让 __subclasshook__ 信服）。当然，只有提供 __subclasshook__ 方法的抽象基类才能这么做。

在自己定义的抽象基类中要不要实现 __subclasshook__ 方法呢？可能不需要。我在 Python 源码中只见到 Sized 这一个抽象基类实现了 __subclasshook__ 方法，而 Sized 只声明了一个特殊方法，因此只用检查这么一个特殊方法。鉴于 __len__ 方法的“特殊性”，我们基本可以确定它能做到该做的事。但是对其他特殊方法和基本的抽象基类来说，很难这么肯定。例如，虽然映射实现了 __len__、__getitem__ 和 __iter__，但是不应该把它们视作 Sequence 的子类型，因为不能使用整数偏移值获取元素，也不能保证元素的顺序。当然，OrderedDict 除外，它保留了插入元素的顺序，但是不支持通过偏移获取元素。

在你我自己编写的抽象基类中实现 __subclasshook__ 方法，可靠性很低。我可不相信随便一个实现或继承了 load、pick、inspect 和 loaded 的类（如 Spam）的行为一定像 Tombola。程序员最好让 Spam 继承 Tombola，至少也要注册（Tombola.register(Spam)），从而确保这一点。当然，自己实现的 __subclasshook__ 方法还可以检查方法签名和其他特性，但我觉得不值得这么做。

11.11　本章小结

本章首先介绍了非正式接口（称为协议）的高度动态本性，然后讲解了抽象基类的静态接口声明，最后指出了抽象基类的动态特性：虚拟子类，以及使用 `__subclasshook__` 方法动态识别子类。

我们首先回顾了 Python 社区对接口的惯常理解。在 Python 的历史中常常出现接口的身影，但它是非正式的，类似于 Smalltalk 的协议，而且在官方文档中，“foo 协议”“foo 接口”和“foo 类对象”这三种措辞是同一个意思。协议风格的接口与继承完全没有关系，实现同一个协议的各个类是相互独立的。在鸭子类型中，接口就是这样的。

通过示例 11-3，我们发现 Python 对序列协议的支持十分深入。如果一个类实现了 `__getitem__` 方法，此外什么也没做，那么 Python 会设法迭代它，而且 `in` 运算符也随之可以使用。随后，我们继续编写第 1 章中的 `FrenchDeck` 示例，还动态添加了一个方法，从而让它支持洗牌。这里用到的是猴子补丁，突出了协议的动态本性。我们再一次见识到，部分实现协议也是有用的：添加可变序列协议中的 `__setitem__` 方法之后，立即就能使用标准库中的 `random.shuffle` 函数。了解现有的协议能让我们充分利用 Python 丰富的标准库。

接下来，Alex Martelli 介绍了“白鹅类型”这个术语，[18] 以此描述一种新的 Python 编程风格。借助“白鹅类型”，可以使用抽象基类明确声明接口，而且类可以子类化抽象基类或使用抽象基类注册（无需在继承关系中确立静态的强链接），宣称它实现了某个接口。

> [18]“白鹅类型”这种说法是 Alex 发明的，这是它第一次出现在书中。

`FrenchDeck2` 示例清楚地展示了显式继承抽象基类的优缺点。继承 `abc.MutableSequence` 后，必须实现 `insert` 和 `__delitem__` 方法，而我们并不需要这两个方法。不过，即便是 Python 新手，只要查看 `FrenchDeck2` 类的源码，就能看出它是可变序列。此外，我们还得到一个额外好处，从 `abc.MutableSequence` 中继承了 11 个方法（其中五个间接继承自 `abc.Sequence`），而且拿来即用。

全面介绍图 11-3 中 `collections.abc` 模块里的各个抽象基类后，我们自己动手从头开始编写了一个抽象基类。PyMOTW.com（Python Module of the Week, http://pymotw.com）网站的创建者 Doug Hellmann 道出了这么做的目的：

> 定义抽象基类之后，各个子类可以实现通用的 API。如果有人不熟悉应用程序的运作方式，却又想使用插件扩展，就可以利用这一功能……[19]

> [19]PyMOTW 网站介绍 abc 模块的页面，“Why use Abstract Base Classes?”一节（https://pymotw.com/2/abc/index.html#why-use-abstract-base-classes）。

定义好 `Tombola` 抽象基类之后，我们创建了三个具体子类，两个继承 `Tombola`，另一个注册为虚拟子类——它们都能通过同一个测试组件。

本章结束之前，我们提到了几个内置类型是如何注册到 `collections.abc` 模块中的抽象基类的。这样，虽然 `memoryview` 没有继承 `abc.Sequence`，`isinstance(memoryview, abc. Sequence)` 的结果也是 `True`。最后，我们探究了 `__subclasshook__` 魔法。这个方法的作用是让抽象基类识别没有注册为子类的类，你可以根据需要做简单的或者复杂的测试——标准库的做法只是检查方法名称。

最后的最后，我要重申 Alex Martelli 的警告：不要自己定义抽象基类，除非你要构建允许用户扩展的框架——然而大多数情况下并非如此。日常使用中，我们与抽象基类的联系应该是创建现有抽象基类的子类，或者使用现有的抽象基类注册。此外，我们可能还会在 `isinstance` 检查中使用抽象基类，但这比继承或注册更少见。需要自己从头编写新抽象基类的情况少之又少。

我使用 Python 15 年了，除了教学示例以外，我只在 Pingo 项目（http://pingo.io）中编写过一个抽象类，即 `Board` 类（https://github.com/garoa/pingo/blob/master/pingo/board.py）。支持单板机和控制器的驱动是 `Board` 的子类，共用相同的接口。就算我把 `pingo.Board` 打造成抽象类，它也并没有继承 `abc.ABC`。[20] 我本打算把 `Board` 定义为抽象基类，但是 Pingo 项目有更重要的事情要做。

> [20]ython 标准库也有这样做的，有些类虽然是抽象的，但是并没有显式地继承 `abc.ABC`。

本章适合使用下面这段话结尾：

> 尽管抽象基类使得类型检查变得更容易了，但不应该在程序中过度使用它。Python 的核心在于它是一门动态语言，它带来了极大的灵活性。如果处处都强制实行类型约束，那么会使代码变得更加复杂，而本不应该如此。我们应该拥抱 Python 的灵活性。[21]
>
> ——David Beazley 和 Brian Jones
> 《Python Cookbook（第 3 版）中文版》

[21]《Python Cookbook（第 3 版）中文版》第 281 页。

或者，像本书技术审校 Leonardo Rochael 所写的：“如果觉得自己想创建新的抽象基类，先试着通过常规的鸭子类型来解决问题。”

11.12　延伸阅读

Beazley 与 Jones 的《Python Cookbook（第 3 版）中文版》有一节（8.12）定义了一个抽象基类。这本书在 Python 3.4 之前撰写，因此他们没有使用现在推荐的句法，即通过继承 `abc.ABC` 声明抽象基类，而是使用 `metaclass` 关键字。除了这个小细节之外，那个秘笈很好地涵盖了抽象基类的主要功能，而且最后还给出了宝贵的意见，即前一节末尾引用的那段话。

Doug Hellmann 写的《Python 标准库》一书中有一章是关于 `abc` 模块的。Doug 创建的 PyMOTW（Python Module of the Week）网站中也有那一章（http://pymotw.com/2/abc/index.html）。这本书和 PyMOTW 网站都针对 Python 2，因此如果你使用 Python 3 的话，必须做些调整。[22] 记住，在 Python 3.4 中，唯一推荐使用的抽象基类方法装饰器是 `@abstractmethod`，其他装饰器已经废弃了。本章小结中引用的关于抽象基类的另一句话出自 Doug 的网站和这本书。

> [22]PyMOTW 网站现在已经是面向 Python 3 了。——编者注

使用抽象基类时，经常会遇到多重继承，而且是不可避免的，因为基本的集合抽象基类（`Sequence`、`Mapping` 和 `Set`）都扩展多个抽象基类（如图 11-3 所示）。第 12 章接着讨论这个话题，那是重要的一章。

“PEP 3119—Introducing Abstract Base Classes”（https://www.python.org/dev/peps/pep-3119）讲解了抽象基类的基本原理，“PEP 3141—A Type Hierarchy for Numbers”（https://www.python.org/dev/peps/pep-3141/）提出了 `numbers` 模块（https://docs.python.org/3/library/numbers.html）中的抽象基类。

Bill Venners 对 Guido van Rossum 的采访“Contracts in Python: A Conversation with Guido van Rossum, Part IV”（http://www.artima.com/intv/pycontract.html）讨论了动态类型的优缺点。

`zope.interface` 包（http://docs.zope.org/zope.interface/）提供了一种声

明接口的方式：检查对象是否实现了接口，注册提供方，然后查询指定接口的提供方。一开始，这个包是 Zope 3 核心的一部分，不过它可以在 Zope 外部使用，而且已经有人这么做了。这个包为大型 Python 项目（如 Twisted、Pyramid 和 Plone）的组件式架构提供了灵活的基础。Lennart Regebro 写的"A Python Component Architecture"一文（https://regebro.wordpress.com/2007/11/16/a-python-component-architecture/）对 `zope.interface` 包做了介绍，Baiju M 还写了一本相关的书——*A Comprehensive Guide to Zope Component Architecture*（http://muthukadan.net/docs/zca.html）。

杂谈

类型提示

2014 年，Python 世界最大的新闻应该是 Guido van Rossum 同意实现可选的静态类型检查，这与检查程序 Mypy（http://www.mypy-lang.org）的做法类似，即使用函数注解实现。这一消息出自 8 月 15 日发表在 Python-ideas 邮件列表中的一个话题，题为"Optional static typing —the crossroads"（https://mail.python.org/pipermail/python-ideas/2014-August/028742.html）。一个月后，"PEP 484—Type Hints"草案（https://www.python.org/dev/peps/pep-0484/）发布了，发起人是 Guido。

这个功能的目的是让程序员在函数定义中使用注解声明参数和返回值的类型，但这是可选的。关键在于"可选"二字。仅当你想得到注解的好处和限制时才需要添加注解，而且可以在一些函数中添加，在另一些函数中不添加。

从表面上看，这与 Microsoft 对 TypeScript（JavaScript 的超集）采取的方式类似，不过 TypeScript 做得更进一步：TypeScript 添加了新的语言结构（如模块、类、显式接口，等等），允许声明变量类型，而且最终编译成常规的 JavaScript。目前来看，Python 的可选静态类型没这么大的雄心。

为了理解这个提案的动机，不能忽略 Guido 在 2014 年 8 月 15 日发送的那封重要邮件中的这段话：

> 我还得做个假设：这个功能主要供 lint 程序、IDE 和文档生成工具使用。这些工具有个共同点：即使类型检查失败了，程序仍能运行。此外，程序中添加的类型不能降低性能（也不能提升性能 :-)）。

因此，这一举动并不像乍一看那么激进。“PEP 484—Type Hints”（https://www.python.org/dev/peps/pep-0484/）提到了“PEP 482—Literature Overview for Type Hints”（https://www.python.org/dev/peps/pep-0482/），后者概述了第三方 Python 工具和其他语言实现类型提示的方式。

不管激进不激进，类型提示都将到来：支持 PEP 484 的 `typing` 模块好像已经纳入 Python 3.5。[23] 根据这个提案的表述和实现方式，可以肯定的是，现有代码不会因为缺少类型提示（或相关的附加物）而无法运行。

最后，PEP 484 明确指出：

> 还要强调一点，Python 依旧是一门动态类型语言，作者从未打算强制要求使用类型提示，甚至不会把它变成约定。

Python 是弱类型语言吗

由于缺少统一的术语，讨论语言类型方面的话题时有时会让人不明其意。有些人（例如扩展阅读中提到的 Bill Venners 对 Guido 的访谈）说 Python 是弱类型语言，把 Python 与 JavaScript 和 PHP 归为一类。讨论类型时，最好考虑两条不同的坐标线。

强类型和弱类型

如果一门语言很少隐式转换类型，说明它是强类型语言；如果经常这么做，说明它是弱类型语言。Java、C++ 和 Python 是强类型语言。PHP、JavaScript 和 Perl 是弱类型语言。

静态类型和动态类型

在编译时检查类型的语言是静态类型语言，在运行时检查类型的语言是动态类型语言。静态类型需要声明类型（有些现代语言使

用类型推导避免部分类型声明）。Fortran 和 Lisp 是最早的两门语言，现在仍在使用，它们分别是静态类型语言和动态类型语言。

强类型能及早发现缺陷。

下面几例体现了弱类型的不足：[24]

```
// 这些是JavaScript代码（在Node.js v0.10.33中做了测试）
'' == '0'    // false
0 == ''      // true
0 == '0'     // true
'' < 0       // false
'' < '0'     // true
```

因为 Python 不会自动在字符串和数字之间强制转换，所以在 Python 3 中，上述 `==` 表达式的结果都是 `False`（保留了 `==` 的意思），而 `<` 比较会抛出 `TypeError`。

静态类型使得一些工具（编译器和 IDE）便于分析代码、找出错误和提供其他服务（优化、重构，等等）。动态类型便于代码重用，代码行数更少，而且能让接口自然成为协议而不提早实行。

综上，Python 是动态强类型语言。“PEP 484—Type Hints”（https://www.python.org/dev/peps/pep-0484/）无法改变这一点，但是 API 作者能够添加可选的类型注解，执行某种静态类型检查。

猴子补丁

猴子补丁的名声不太好。如果滥用，会导致系统难以理解和维护。补丁通常与目标紧密耦合，因此很脆弱。另一个问题是，打了猴子补丁的两个库可能相互牵绊，因为第二个库可能撤销了第一个库的补丁。

不过猴子补丁也有它的作用，例如可以在运行时让类实现协议。适配器设计模式通过实现全新的类解决这种问题。

为 Python 打猴子补丁不难，但是有些局限。与 Ruby 和 JavaScript

不同，Python 不允许为内置类型打猴子补丁。其实我觉得这是优点，因为这样可以确保 `str` 对象的方法始终是那些。这一局限能减少外部库打的补丁有冲突的概率。

Java、Go 和 Ruby 的接口

从 C++ 2.0（1989 年发布）起，这门语言开始使用抽象类指定接口。Java 的设计者选择不支持类的多重继承，这排除了使用抽象类作为接口规范的可能性，因为一个类通常会实现多个接口。但是，Java 的设计者添加了 `interface` 这个语言结构，而且允许一个类实现多个接口——这是一种多重继承。以更为明确的方式定义接口是 Java 的一大贡献。在 Java 8 中，接口可以提供方法实现，这叫默认方法（https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html）。有了这个功能，Java 的接口与 C++ 和 Python 中的抽象类更像了。

Go 语言采用的方式完全不同。首先，Go 不支持继承。我们可以定义接口，但是无需（其实也不能）明确地指出某个类型实现了某个接口。编译器能自动判断。因此，考虑到接口在编译时检查，但是真正重要的是实现了什么类型，Go 语言可以说是具有“静态鸭子类型”。

与 Python 相比，对 Go 来说就好像每个抽象基类都实现了 `__subclasshook__` 方法，它会检查函数的名称和签名，而我们自己从不需要继承或注册抽象基类。如果想让 Python 更像 Go，可以对所有函数参数做类型检查。Python 提供了部分基础设施（参见 5.9 节）。Guido 说过，他不介意使用注解做类型检查，至少在辅助工具中可以这么做。详情参阅第 5 章的“杂谈”。

Ruby 程序员是鸭子类型的坚定拥护者，而且 Ruby 没有声明接口或抽象类的正式方式，只能像 Python 2.6 之前的版本那样做，即在方法的定义体中抛出 `NotImplementedError`，以此表明方法是抽象的，用户必须在子类中实现。

不过，2014 年 9 月，Ruby 之父松本行弘在日本举办的 Ruby Kaigi（最重要的 Ruby 大会之一，每年举办）中做了一场主题演讲，他透露说，Ruby 未来可能会支持静态类型。目前我还没看到相关报道，但是根据 Godfrey Chan 的博客文章“Ruby Kaigi 2014:

Day 2”（http://brewhouse.io/blog/2014/09/19/ruby-kaigi-2014-day-2），松本行弘关注的似乎是函数注解。他甚至还提到了 Python 的函数注解。

在没有抽象基类向类型系统添加结构，以及不丧失灵活性的情况下，我不知道函数注解有什么用。因此，Ruby 未来可能还会支持正式接口。

我相信，Python 的抽象基类在 `register` 函数和 `__subclasshook__` 方法的协助下能把正式接口带入这门语言，而且不失去动态类型的优势。

或许，鹅正在赶超鸭子。

接口中的隐喻和习惯用法

隐喻能打破壁垒，让人更易于理解。使用“栈”和“队列”描述基本的数据类型就有这样的功效：这两个词清楚地道出了添加或删除元素的方式。另一方面，Alan Cooper 在《交互设计精髓（第 4 版）》中写道：

> 严格奉行隐喻设计毫无必要，却把界面死死地与物理世界的运行机制捆绑在一起。

他说的是用户界面，但对 API 同样适用。不过 Cooper 同意，当“真正合适的”隐喻“正中下怀”时，可以使用隐喻（他用的词是“正中下怀”，因为合适的隐喻可遇不可求）。我觉得本章用宾果机作比喻是合适的，我相信自己。

我读过不少 UI 设计方面的书，《交互设计精髓》是最好的。我从 Cooper 的书中学到的最宝贵的知识是，不把隐喻当作设计范式，而代之以“习惯用法的界面”。前面说过，Cooper 说的不是 API，但是我越深入思考他的观点，越觉得可以将之运用到 Python 中。Python 语言的基本协议就是 Cooper 所说的“习惯用法”。知道“序列”是什么之后，可以把这些知识应用到不同的场合。这正是本书的主要目的：着重讲解这门语言的基本惯用法，让你的代码简洁、高效且可读，把你打造成熟练的 Python 程序员。

[23]现在，`typing` 模块已经纳入 Python 3.5。——编者注

[24]改编自 *JavaScript: The Good Parts*（Douglas Crockford 著）附录 B 第 109 页给出的示例。

第 12 章　继承的优缺点

> （我们）推出继承的初衷是让新手顺利使用只有专家才能设计出来的框架。[1]
>
> ——Alan Kay
> “The Early History of Smalltalk”

> [1]Alan Kay,“The Early History of Smalltalk,”in SIGPLAN Not. 28, 3 (March 1993), 69-95. 网上也有这篇文章（http://propella.sakura.ne.jp/earlyHistoryST/EarlyHistoryST.html）。感谢我的朋友 Christiano Anderson 在我写这一章时告诉我这篇参考文献。

本章探讨继承和子类化，重点是说明对 Python 而言尤为重要的两个细节：

- 子类化内置类型的缺点
- 多重继承和方法解析顺序

很多人觉得多重继承得不偿失。不支持多重继承的 Java 显然没有什么损失，C++ 对多重继承的滥用伤害了很多人，这可能还坚定了使用 Java 的决心。

然而，Java 的巨大成功和广泛影响，也导致很多刚接触 Python 的程序员没怎么见过真实的代码使用多重继承。鉴于此，我们将不再举简单的示例，而是通过两个重要的 Python 项目探讨多重继承，这两个项目是 GUI 工具包 Tkinter 和 Web 框架 Django。

我们将首先分析子类化内置类型的问题。本章余下的内容则探讨多重继承，我们将分析案例，并讨论构建类层次结构方面好的做法和不好的做法。

12.1　子类化内置类型很麻烦

在 Python 2.2 之前，内置类型（如 `list` 或 `dict`）不能子类化。在 Python 2.2 之后，内置类型可以子类化了，但是有个重要的注意事项：内置类型（使用 C 语言编写）不会调用用户定义的类覆盖的特殊方法。

PyPy 的文档使用简明扼要的语言描述了这个问题，见于“Differences between PyPy and CPython”中“Subclasses of built-in types”一节（http://pypy.readthedocs.io/en/latest/cpython_differences.html#subclasses-of-built-in-types）：

> 至于内置类型的子类覆盖的方法会不会隐式调用，CPython 没有制定官方规则。基本上，内置类型的方法不会调用子类覆盖的方法。例如，`dict` 的子类覆盖的 `__getitem__()` 方法不会被内置类型的 `get()` 方法调用。

示例 12-1 说明了这个问题。

示例 **12-1**　内置类型 `dict` 的 `__init__` 和 `__update__` 方法会忽略我们覆盖的 `__setitem__` 方法

```
>>> class DoppelDict(dict):
...     def __setitem__(self, key, value):
...         super().__setitem__(key, [value] * 2)  # ❶
...
>>> dd = DoppelDict(one=1)  # ❷
>>> dd
{'one': 1}
>>> dd['two'] = 2  # ❸
>>> dd
{'one': 1, 'two': [2, 2]}
>>> dd.update(three=3)  # ❹
>>> dd
{'three': 3, 'one': 1, 'two': [2, 2]}
```

❶ `DoppelDict.__setitem__` 方法会重复存入的值（只是为了提供易于观察的效果）。它把职责委托给超类。

❷ 继承自 dict 的 __init__ 方法显然忽略了我们覆盖的 __setitem__ 方法：'one' 的值没有重复。

❸ [] 运算符会调用我们覆盖的 __setitem__ 方法，按预期那样工作：'two' 对应的是两个重复的值，即 [2, 2]。

❹ 继承自 dict 的 update 方法也不使用我们覆盖的 __setitem__ 方法：'three' 的值没有重复。

原生类型的这种行为违背了面向对象编程的一个基本原则：始终应该从实例（self）所属的类开始搜索方法，即使在超类实现的类中调用也是如此。在这种糟糕的局面中，__missing__ 方法（参见 3.4.2 节）却能按预期方式工作，不过这只是特例。

不只实例内部的调用有这个问题（self.get() 不调用 self.__getitem__()），内置类型的方法调用的其他类的方法，如果被覆盖了，也不会被调用。示例 12-2 是一个例子，改编自 PyPy 文档中的示例（http://pypy.readthedocs.io/en/latest/cpython_differences.html#subclasses-of-built-in-types）。

示例 **12-2**　dict.update 方法会忽略 AnswerDict.__getitem__ 方法

```
>>> class AnswerDict(dict):
...     def __getitem__(self, key):  # ❶
...         return 42
...
>>> ad = AnswerDict(a='foo')  # ❷
>>> ad['a']  # ❸
42
>>> d = {}
>>> d.update(ad)  # ❹
>>> d['a']  # ❺
'foo'
>>> d
{'a': 'foo'}
```

❶ 不管传入什么键，AnswerDict.__getitem__ 方法始终返回 42。

❷ ad 是 AnswerDict 的实例，以 ('a', 'foo') 键值对初始化。

❸ ad['a'] 返回 42，这与预期相符。

❹ d 是 dict 的实例，使用 ad 中的值更新 d。

❺ dict.update 方法忽略了 AnswerDict.__getitem__ 方法。

直接子类化内置类型（如 dict、list 或 str）容易出错，因为内置类型的方法通常会忽略用户覆盖的方法。不要子类化内置类型，用户自己定义的类应该继承 collections 模块（http://docs.python.org/3/library/collections.html）中的类，例如 UserDict、UserList 和 UserString，这些类做了特殊设计，因此易于扩展。

如果不子类化 dict，而是子类化 collections.UserDict，示例 12-1 和示例 12-2 中暴露的问题便迎刃而解了。参见示例 12-3。

示例 **12-3**　DoppelDict2 和 AnswerDict2 能像预期那样使用，因为它们扩展的是 UserDict，而不是 dict

```
>>> import collections
>>>
>>> class DoppelDict2(collections.UserDict):
...     def __setitem__(self, key, value):
...         super().__setitem__(key, [value] * 2)
...
>>> dd = DoppelDict2(one=1)
>>> dd
{'one': [1, 1]}
>>> dd['two'] = 2
>>> dd
{'two': [2, 2], 'one': [1, 1]}
>>> dd.update(three=3)
>>> dd
{'two': [2, 2], 'three': [3, 3], 'one': [1, 1]}
>>>
>>> class AnswerDict2(collections.UserDict):
...     def __getitem__(self, key):
...         return 42
...
```

```
>>> ad = AnswerDict2(a='foo')
>>> ad['a']
42
>>> d = {}
>>> d.update(ad)
>>> d['a']
42
>>> d
{'a': 42}
```

为了衡量子类化内置类型所需的额外工作量，我做了个实验，重写了示例 3-8 中的 `StrKeyDict` 类。原始版继承自 `collections.UserDict`，而且只实现了三个方法：`__missing__`、`__contains__` 和 `__setitem__`。在实验中，`StrKeyDict` 直接子类化 `dict`，而且也实现了那三个方法，不过根据存储数据的方式稍微做了调整。可是，为了让实验版通过原始版的测试组件，还要实现 `__init__`、`get` 和 `update` 方法，因为继承自 `dict` 的版本拒绝与覆盖的 `__missing__`、`__contains__` 和 `__setitem__` 方法合作。示例 3-8 中那个 `UserDict` 子类有 16 行代码，而实验的 `dict` 子类有 37 行代码。[2]

> [2]如果好奇，实验版在本书代码仓库（https://github.com/fluentpython/example-code）里的 strkeydict_ dictsub.py 文件中。

综上，本节所述的问题只发生在 C 语言实现的内置类型内部的方法委托上，而且只影响直接继承内置类型的用户自定义类。如果子类化使用 Python 编写的类，如 `UserDict` 或 `MutableMapping`，就不会受此影响。[3]

> [3]顺便说一下，在这方面，PyPy 的行为比 CPython“正确”，不过会导致微小的差异。详情参见“Differences between PyPy and CPython”（http://pypy.readthedocs.io/en/latest/cpython_differences.html#subclasses-of-built-in-types）。

与继承，尤其是多重继承有关的另一个问题是：如果同级别的超类定义了同名属性，Python 如何确定使用哪个？下一节解答。

12.2 多重继承和方法解析顺序

任何实现多重继承的语言都要处理潜在的命名冲突，这种冲突由不相关的祖先类实现同名方法引起。这种冲突称为“菱形问题”，如图 12-1 和示例 12-4 所示。

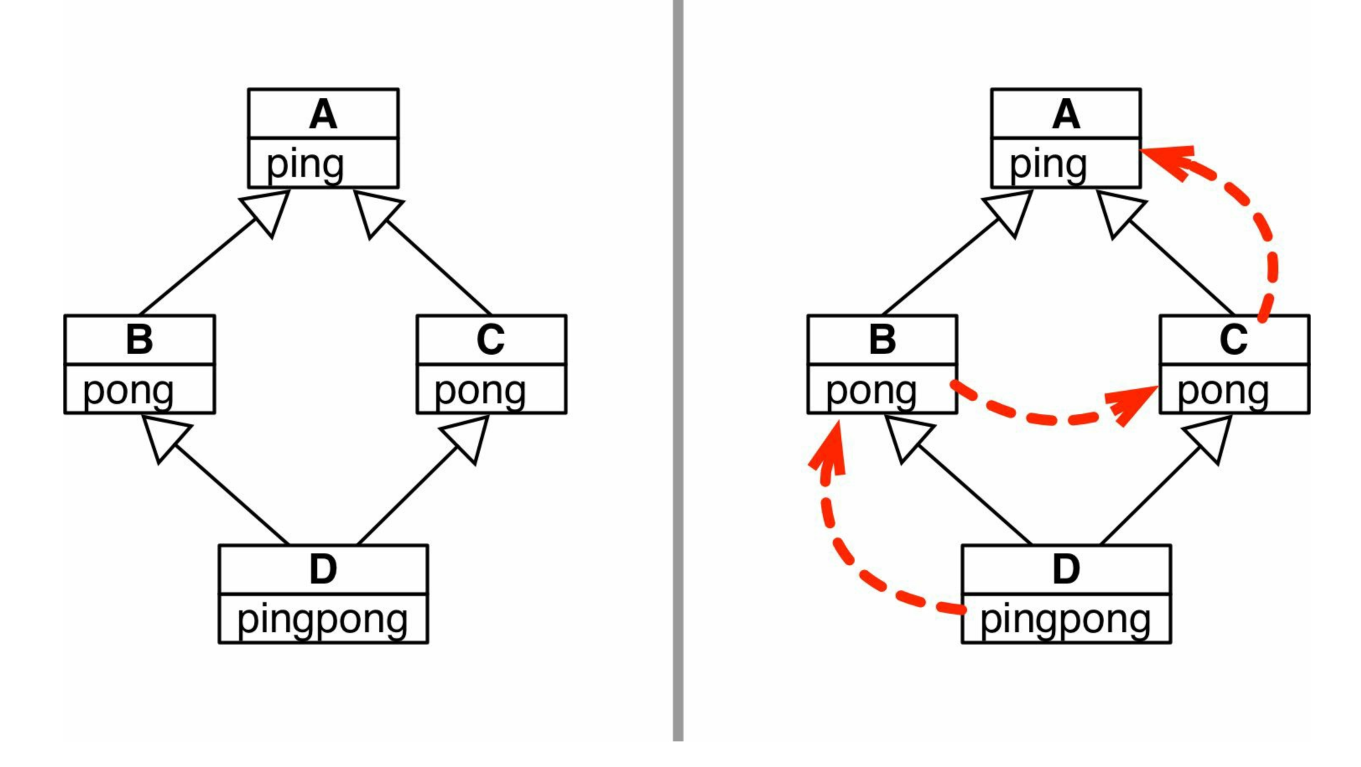


图 **12-1**：（左）说明“菱形问题”的 **UML** 类图；（右）虚线箭头是示例 **12-4** 使用的方法解析顺序

示例 **12-4**　diamond.py：图 12-1 中的 A、B、C 和 D 四个类

```
class A:
    def ping(self):
        print('ping:', self)


class B(A):
    def pong(self):
        print('pong:', self)


class C(A):
    def pong(self):
        print('PONG:', self)


class D(B, C):

    def ping(self):
        super().ping()
```

```
        print('post-ping:', self)

    def pingpong(self):
        self.ping()
        super().ping()
        self.pong()
        super().pong()
        C.pong(self)
```

注意，B 和 C 都实现了 pong 方法，二者之间唯一的区别是，C.pong 方法输出的是大写的 PONG。

在 D 的实例上调用 d.pong() 方法的话，运行的是哪个 pong 方法呢？在 C++ 中，程序员必须使用类名限定方法调用来避免这种歧义。Python 也能这么做，如示例 12-5 所示。

示例 **12-5**　在 D 实例上调用 pong 方法的两种方式

```
>>> from diamond import *
>>> d = D()
>>> d.pong()  # ❶
pong: <diamond.D object at 0x10066c278>
>>> C.pong(d)  # ❷
PONG: <diamond.D object at 0x10066c278>
```

❶ 直接调用 d.pong() 运行的是 B 类中的版本。

❷ 超类中的方法都可以直接调用，此时要把实例作为显式参数传入。

Python 能区分 d.pong() 调用的是哪个方法，是因为 Python 会按照特定的顺序遍历继承图。这个顺序叫方法解析顺序（Method Resolution Order，MRO）。类都有一个名为 __mro__ 的属性，它的值是一个元组，按照方法解析顺序列出各个超类，从当前类一直向上，直到 object 类。D 类的 __mro__ 属性如下（如图 12-1 所示）：

```
>>> D.__mro__
(<class 'diamond.D'>, <class 'diamond.B'>, <class 'diamond.C'>,
<class 'diamond.A'>, <class 'object'>)
```

若想把方法调用委托给超类，推荐的方式是使用内置的 super() 函

数。在 Python 3 中，这种方式变得更容易了，如示例 12-4 中 D 类的 pingpong 方法所示。[4] 然而，有时可能需要绕过方法解析顺序，直接调用某个超类的方法——这样做有时更方便。例如，D.ping 方法可以这样写：

> [4]在 Python 2 中，要把 D.pingpong 方法的第二行从 super().ping() 改成 super(D, self).ping()。

```
    def ping(self):
        A.ping(self)  # 而不是super().ping()
        print('post-ping:', self)
```

注意，直接在类上调用实例方法时，必须显式传入 self 参数，因为这样访问的是未绑定方法（unbound method）。

然而，使用 super() 最安全，也不易过时。调用框架或不受自己控制的类层次结构中的方法时，尤其适合使用 super()。使用 super() 调用方法时，会遵守方法解析顺序，如示例 12-6 所示。

示例 **12-6**　使用 super() 函数调用 ping 方法（源码在示例 12-4 中）

```
>>> from diamond import D
>>> d = D()
>>> d.ping()  # ❶
ping: <diamond.D object at 0x10cc40630>  # ❷
post-ping: <diamond.D object at 0x10cc40630>  # ❸
```

❶ D 类的 ping 方法做了两次调用。

❷ 第一个调用是 super().ping()；super 函数把 ping 调用委托给 A 类；这一行由 A.ping 输出。

❸ 第二个调用是 print('post-ping:', self)，输出的是这一行。

下面来看在 D 实例上调用 pingpong 方法得到的结果，如示例 12-7 所示。

示例 **12-7** pingpong 方法的 5 个调用（源码在示例 12-4 中）

```
>>> from diamond import D
>>> d = D()
>>> d.pingpong()
ping: <diamond.D object at 0x10bf235c0>  # ❶
post-ping: <diamond.D object at 0x10bf235c0>
ping: <diamond.D object at 0x10bf235c0>  # ❷
pong: <diamond.D object at 0x10bf235c0>  # ❸
pong: <diamond.D object at 0x10bf235c0>  # ❹
PONG: <diamond.D object at 0x10bf235c0>  # ❺
```

❶ 第一个调用是 `self.ping()`，运行的是 D 类的 ping 方法，输出这一行和下一行。

❷ 第二个调用是 `super().ping()`，跳过 D 类的 ping 方法，找到 A 类的 ping 方法。

❸ 第三个调用是 `self.pong()`，根据 `__mro__`，找到的是 B 类实现的 pong 方法。

❹ 第四个调用是 `super().pong()`，也根据 `__mro__`，找到 B 类实现的 pong 方法。

❺ 第五个调用是 `C.pong(self)`，忽略 mro，找到的是 C 类实现的 pong 方法。

方法解析顺序不仅考虑继承图，还考虑子类声明中列出超类的顺序。也就是说，如果在 diamond.py 文件（见示例 12-4）中把 D 类声明为 `class D(C, B):`，那么 D 类的 `__mro__` 属性就会不一样：先搜索 C 类，再搜索 B 类。

分析类时，我经常在交互式控制台中查看 `__mro__` 属性。示例 12-8 中是一些常用类的方法搜索顺序。

示例 **12-8** 查看几个类的 `__mro__` 属性

```
>>> bool.__mro__ ❶
(<class 'bool'>, <class 'int'>, <class 'object'>)
>>> def print_mro(cls): ❷
```

```
...     print(', '.join(c.__name__ for c in cls.__mro__))
...
>>> print_mro(bool)
bool, int, object
>>> from frenchdeck2 import FrenchDeck2
>>> print_mro(FrenchDeck2) ❸
FrenchDeck2, MutableSequence, Sequence, Sized, Iterable, Container, object
>>> import numbers
>>> print_mro(numbers.Integral) ❹
Integral, Rational, Real, Complex, Number, object
>>> import io ❺
>>> print_mro(io.BytesIO)
BytesIO, _BufferedIOBase, _IOBase, object
>>> print_mro(io.TextIOWrapper)
TextIOWrapper, _TextIOBase, _IOBase, object
```

❶ `bool` 从 `int` 和 `object` 中继承方法和属性。

❷ `print_mro` 函数使用更紧凑的方式显示方法解析顺序。

❸ `FrenchDeck2` 类的祖先包含 `collections.abc` 模块中的几个抽象基类。

❹ 这些是 `numbers` 模块提供的几个数字抽象基类。

❺ `io` 模块中有抽象基类（名称以 `...Base` 后缀结尾）和具体类，如 `BytesIO` 和 `TextIOWrapper`。`open()` 函数返回的对象属于这些类型，具体要根据模式参数而定。



方法解析顺序使用 C3 算法计算。Michele Simionato 的论文“The Python 2.3 Method Resolution Order”（https://www.python.org/download/releases/2.3/mro/）对 Python 方法解析顺序使用的 C3 算法做了权威论述。如果对方法解析顺序的细节感兴趣，可以阅读延伸阅读中给出的资料。不用过分担心，C3 算法不难理解，Simionato 写道：

> ……除非大量使用多重继承，或者继承关系不同寻常，否则不用了解 C3 算法，因此也不用阅读这篇论文。

结束对方法解析顺序的讨论之前，我们来看看图 12-2。这幅图展示了 Python 标准库中 GUI 工具包 Tkinter 复杂的多重继承图。研究这幅图时，要从底部的 `Text` 类开始。这个类全面实现了多行可编辑文本小组件，它自身有丰富的功能，不过也从其他类继承了很多方法。左边是常规的 UML 类图。右边加入了一些箭头，表示方法解析顺序。使用示例 12-8 中定义的便利函数 `print_mro` 得到的输出如下：

```
>>> import tkinter
>>> print_mro(tkinter.Text)
Text, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, YView, object
```

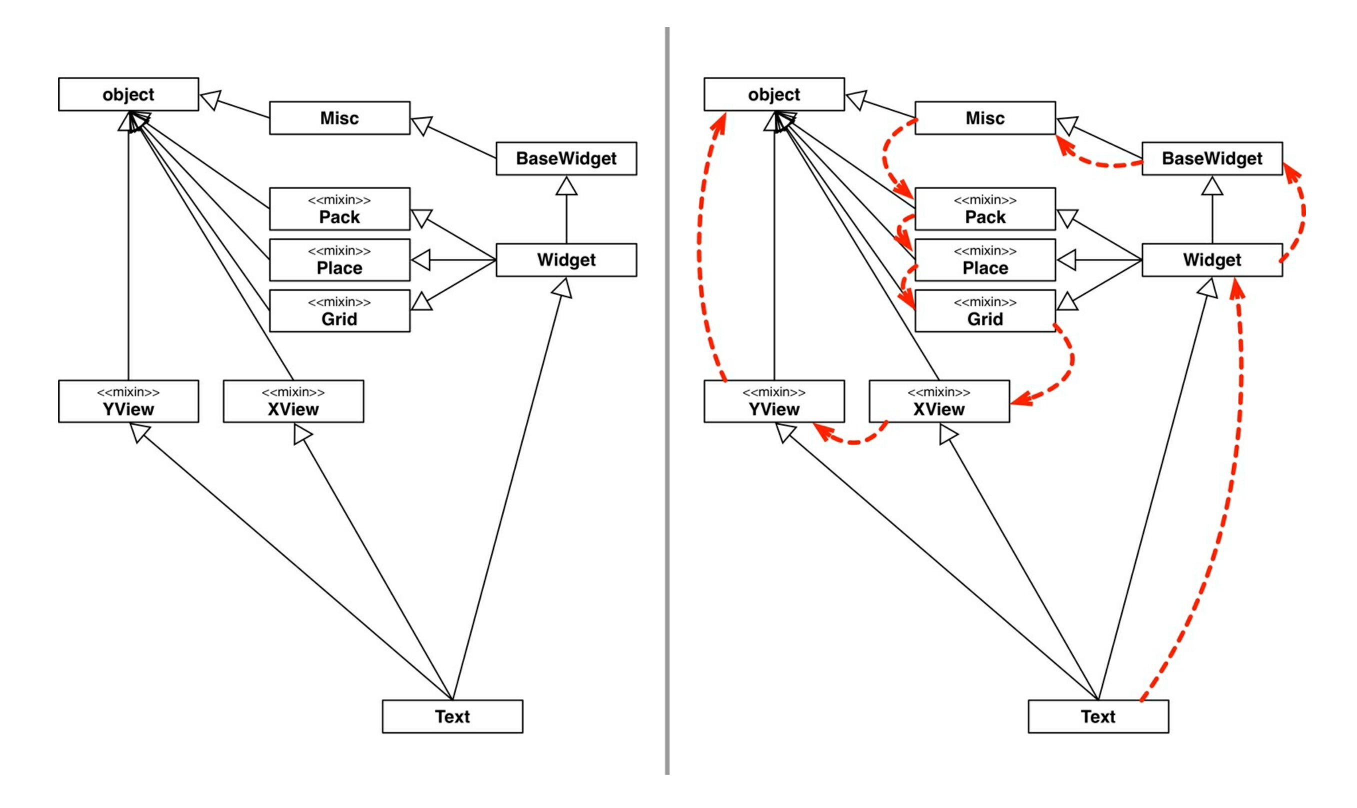


图 **12-2**：（左）**Tkinter** 中 `Text` 小组件类及其超类的 **UML** 类图；（右）使用虚线箭头表示 `Text.__mro__`

下一节以真实框架为例说明多重继承的优缺点。

12.3　多重继承的真实应用

多重继承能发挥积极作用。《设计模式：可复用面向对象软件的基础》一书中的适配器模式用的就是多重继承，因此使用多重继承肯定没有错（那本书中的其他 22 个设计模式都使用单继承，因此多重继承显然不是灵丹妙药）。

在 Python 标准库中，最常使用多重继承的是 `collections.abc` 包。这没什么问题，毕竟连 Java 都支持接口的多重继承，而抽象基类就是接口声明，只不过它可以提供具体方法的实现。[5]

> [5]前面说过，Java 8 也允许提供方法实现。这个新功能在官方的 Java 教程中叫默认方法（https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html）。

在标准库中，GUI 工具包 Tkinter（`tkinter` 模块是 Tcl/Tk 的 Python 接口，https://docs.python.org/3/library/tkinter.html）把多重继承用到了极致。图 12-2 中展示的方法解析顺序是 Tkinter 小组件层次结构的一部分，图 12-3 则列出了 `tkinter` 基包中的全部小组件类（`tkinter.ttk` 子包中还有一些，https://docs.python.org/3/library/tkinter.ttk.html）。

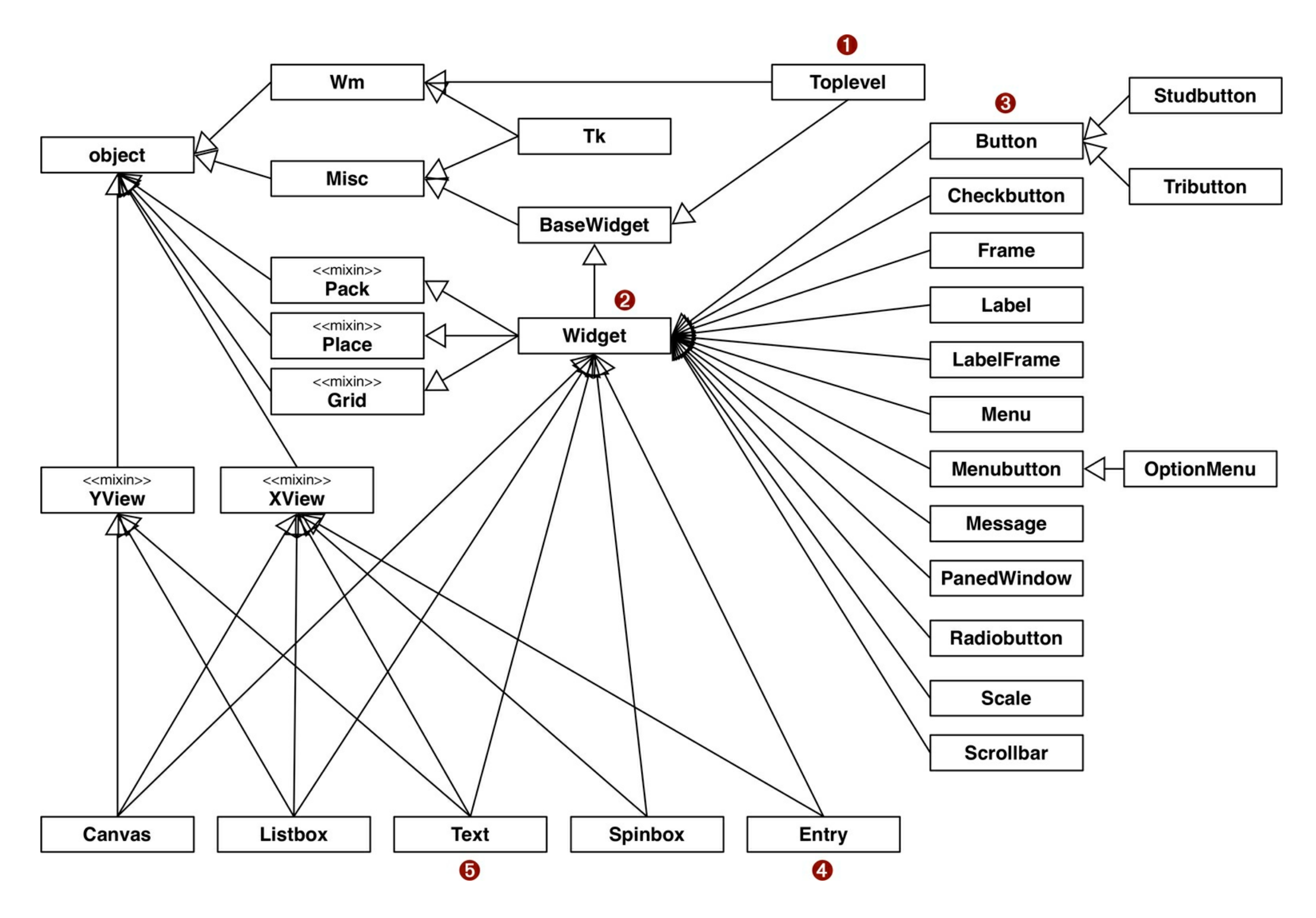


图 **12-3**：**Tkinter GUI** 类层次结构的 **UML** 简图；使用 **«mixin»** 标记的类通过多重继承为其他类提供具体方法

写作本书时，Tkinter 已经 20 岁了，不能代表当下的最佳实践。但是，它却能表明当没有意识到多重继承的缺点时，程序员是如何使用多重继承的。下一节讨论一些好的做法时，会把 Tkinter 作为反面教材。

来看图 12-3 中的几个类。

❶ `Toplevel`：表示 Tkinter 应用程序中顶层窗口的类。

❷ `Widget`：窗口中所有可见对象的超类。

❸ `Button`：普通的按钮小组件。

❹ `Entry`：单行可编辑文本字段。

❺ `Text`：多行可编辑文本字段。

这几个类的方法解析顺序如下，这些输出使用示例 12-8 中定义的 `print_mro` 函数得到：

```
>>> import tkinter
>>> print_mro(tkinter.Toplevel)
Toplevel, BaseWidget, Misc, Wm, object
>>> print_mro(tkinter.Widget)
Widget, BaseWidget, Misc, Pack, Place, Grid, object
>>> print_mro(tkinter.Button)
Button, Widget, BaseWidget, Misc, Pack, Place, Grid, object
>>> print_mro(tkinter.Entry)
Entry, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, object
>>> print_mro(tkinter.Text)
Text, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, YView, object
```

在类之间的关系方面有几点要注意。

- `Toplevel` 是所有图形类中唯一没有继承 `Widget` 的，因为它是顶层窗口，行为不像小组件，例如不能依附到窗口或窗体上。`Toplevel` 继承自 `Wm`，后者提供直接访问宿主窗口管理器的函数，例如设置窗口标题和配置窗口边框。

- `Widget` 直接继承自 `BaseWidget`，还继承了 `Pack`、`Place` 和 `Grid`。后三个类是几何管理器，负责在窗口或窗体中排布小组件。各个类封装了不同的布局策略和小组件位置 API。

- `Button` 与大多数小组件一样，只是 `Widget` 的子代，也间接继承 `Misc`，后者为各个小组件提供了大量方法。

- `Entry` 是 `Widget` 和 `XView` 的子类，后者实现横向滚动。

- `Text` 是 `Widget`、`XView` 和 `YView` 的子类，后者提供纵向滚动功能。

下面将讨论多重继承一些好的做法，看看 Tkinter 有没有践行。

12.4　处理多重继承

> ……我们需要一种更好的、全新的继承理论（目前仍是如此）。例如，继承和实例化（一种继承方式）混淆了语用（比如为了节省空间而重构代码）和语义（用途太多了，比如特殊化、普遍化、形态，等等）。
>
> ——Alan Kay
> “The Early History of Smalltalk”

如 Alan Kay 所言，继承有很多用途，而多重继承增加了可选方案和复杂度。使用多重继承容易得出令人费解和脆弱的设计。我们还没有完整的理论，下面是避免把类图搅乱的一些建议。

01. 把接口继承和实现继承区分开

    使用多重继承时，一定要明确一开始为什么创建子类。主要原因可能有：

    - 继承接口，创建子类型，实现“是什么”关系
    - 继承实现，通过重用避免代码重复

    其实这两条经常同时出现，不过只要可能，一定要明确意图。通过继承重用代码是实现细节，通常可以换用组合和委托模式。而接口继承则是框架的支柱。

02. 使用抽象基类显式表示接口

    现代的 Python 中，如果类的作用是定义接口，应该明确把它定义为抽象基类。Python 3.4 及以上的版本中，我们要创建 `abc.ABC` 或其他抽象基类的子类（如果想支持较旧的 Python 版本，参见 11.7.1 节）。

03. 通过混入重用代码

如果一个类的作用是为多个不相关的子类提供方法实现，从而实现重用，但不体现“是什么”关系，应该把那个类明确地定义为混入类（mixin class）。从概念上讲，混入不定义新类型，只是打包方法，便于重用。混入类绝对不能实例化，而且具体类不能只继承混入类。混入类应该提供某方面的特定行为，只实现少量关系非常紧密的方法。

04. 在名称中明确指明混入

因为在 Python 中没有把类声明为混入的正规方式，所以强烈推荐在名称中加入 `...Mixin` 后缀。Tkinter 没有采纳这个建议，如果采纳的话，`XView` 会变成 `XViewMixin`，`Pack` 会变成 `PackMixin`，图 12-3 中所有使用 «mixin» 标记的类都应该这么做。

05. 抽象基类可以作为混入，反过来则不成立

抽象基类可以实现具体方法，因此也可以作为混入使用。不过，抽象基类会定义类型，而混入做不到。此外，抽象基类可以作为其他类的唯一基类，而混入决不能作为唯一的超类，除非继承另一个更具体的混入——真实的代码很少这样做。

抽象基类有个局限是混入没有的：抽象基类中实现的具体方法只能与抽象基类及其超类中的方法协作。这表明，抽象基类中的具体方法只是一种便利措施，因为这些方法所做的一切，用户调用抽象基类中的其他方法也能做到。

06. 不要子类化多个具体类

具体类可以没有，或最多只有一个具体超类。[6] 也就是说，具体类的超类中除了这一个具体超类之外，其余的都是抽象基类或混入。例如，在下述代码中，如果 `Alpha` 是具体类，那么 `Beta` 和 `Gamma` 必须是抽象基类或混入：

```
class MyConcreteClass(Alpha, Beta, Gamma):
    """这是一个具体类，可以实例化。"""
    # ……更多代码……
```

07. 为用户提供聚合类

如果抽象基类或混入的组合对客户代码非常有用，那就提供一个类，使用易于理解的方式把它们结合起来。Grady Booch 把这种类称为聚合类（aggregate class）。[7]

例如，下面是 tkinter.Widget 类的完整代码（https://hg.python.org/cpython/file/3.4/Lib/tkinter/__init__.py#l2141）：

```
class Widget(BaseWidget, Pack, Place, Grid):
    """Internal class.

    Base class for a widget which can be positioned with the
    geometry managers Pack, Place or Grid."""
    pass
```

Widget 类的定义体是空的，但是这个类提供了有用的服务：把四个超类结合在一起，这样需要创建新小组件的用户无需记住全部混入，也不用担心声明 class 语句时有没有遵守特定的顺序。Django 中的 ListView 类是更好的例子，稍后在 12.5 节讨论。

08. “优先使用对象组合，而不是类继承”

这句话引自《设计模式：可复用面向对象软件的基础》一书，[8] 这是我能提供的最佳建议。熟悉继承之后，就太容易过度使用它了。出于对秩序的诉求，我们喜欢按整洁的层次结构放置物品，程序员更是乐此不疲。

然而，优先使用组合能让设计更灵活。例如，对 tkinter.Widget 类来说，它可以不从全部几何管理器中继承方法，而是在小组件实例中维护一个几何管理器引用，然后通过它调用方法。毕竟，小组件“不是”几何管理器，但是可以通过委托使用相关的服务。这样，我们可以放心添加新的几何管理器，不必担心会触动小组件类的层次结构，也不必担心名称冲突。即便是单继承，这个原则也能提升灵活性，因为子类化是一种紧耦合，而且较高的继承树容易倒。

组合和委托可以代替混入，把行为提供给不同的类，但是不能取代接口继承去定义类型层次结构。

[6]在“水禽和抽象基类”中，Alex Martelli 提到 Scott Meyer 写的 *More Effective C++* 一书，他做得

更绝，“将非尾端类设计为抽象类”（即，具体类根本不应该有具体超类）。

[7]“如果一个类的结构主要继承自混入，自身没有添加结构或行为，那么这样的类称为聚合类。”Grady Booch et al., *Object Oriented Analysis and Design, 3E* (Addison-Wesley, 2007), p. 109.

[8]《设计模式：可复用面向对象软件的基础》第 13 页。

接下来，我们将从这些建议入手分析 Tkinter。

Tkinter好的、不好的和令人厌恶的方面

记住一点，自 1994 年发布的 Python 1.1 起，Tkinter 就在标准库中了。Tkinter 的底层是 Tcl 语言优秀的 GUI 工具包 Tk。Tcl/Tk 组合原本不是面向对象的，因此 Tk API 基本上就是一堆函数。尽管没有使用面向对象方式实现，但是这个工具包的理念极具面向对象思想。

前几节给出的建议 Tkinter 大都没有采用，不过第 7 点是个例外。但是 Tkinter 做得并不好，因为使用组合模式把几何管理器集成到 `Widget` 中更好，如第 8 点所述。

`tkinter.Widget` 类的文档字符串开头说它是“内部类”。这或许表明 `Widget` 应该定义为抽象基类。`Widget` 自身虽然没有方法，但是它定义了接口。它传达的意思是：“每个 Tkinter 小组件都会提供基本的方法（`__init__`、`destroy`，以及众多 Tk API 函数），此外还会提供三个几何管理器中的全部方法。”你可以不同意这是定义接口的好方式（太宽泛了），但是这样确实能定义接口，`Widget` 就把接口“定义”为超类接口的联合。

封装 GUI 应用逻辑的 `Tk` 类继承自 `Wm` 和 `Misc`，这两个类既不是抽象类，也不是混入（`Wm` 不算是混入，因为 `TopLevel` 的超类只有它一个）。`Misc` 类的名称本身明显是代码异味。 `Misc` 有 100 多个方法，而且所有小组件类都继承它。为什么每个小组件都要处理剪切板、文本选择和计时器等？我们可能不能把文本粘贴到按钮上，也不能选择滚动条里的文字。 `Misc` 应该拆分成几个专门的混入类，而且不是所有小组件都应该继承这些混入。

说实在的，作为 Tkinter 的用户，你根本不用知道或使用多重继承。那些都是隐藏起来的实现细节，你在自己的代码中只需实例化或子类化小组件类。不过，如果你想查找自己需要的方法，在控制台中输入 `dir(tkinter.Button)`，你会发现列出了 214 个属性，[9] 此时你就是多重继承的受害者。

> [9]目前的版本中只有 209 个属性。——编者注

除了这些问题，Tkinter 还是稳定而灵活的，未必那么不堪。陈旧（和默认）的 Tk 小组件没有考虑现代的用户界面，但是 Python 3.1（2009 年发布）提供了 `tkinter.ttk` 包，这个包提供的小组件很精美，外观同原生的一样，开发出的 GUI 应用也更专业。此外，有些陈旧的小组件，如 `Canvas` 和 `Text`，功能异常强大。只需少量代码，就能把一个 `Canvas` 对象打造成简单的拖拽绘图应用。如果你对 GUI 编程感兴趣，Tkinter 和 Tcl/Tk 绝对值得一看。

然而，我们的主题不是 GUI 编程，而是多重继承的运用。显式使用混入类的现代示例在 Django 中可以找到。

12.5　一个现代示例：Django通用视图中的混入

> 阅读本节不需要掌握 Django 知识。我只是使用这个框架的一小部分为例说明多重继承的运用，我会尽量给出所需的全部背景知识，而且假设你使用其他语言或框架做过服务器端 Web 开发。

在 Django 中，视图是可调用的对象，它的参数是表示 HTTP 请求的对象，返回值是一个表示 HTTP 响应的对象。我们要关注的是这些响应对象。响应可以是简单的重定向，没有主体内容，也可以是复杂的内容，如在线商店的目录页面，它使用 HTML 模板渲染，列出多个货品，而且有购买按钮和详情页面链接。

起初，Django 提供的是一系列函数，这叫通用视图，实现常见的用例。例如，很多网站都需要展示搜索结果，里面包含很多项目，分成多页，而且各个项目会链接到详细信息页面。在 Django 中，这种需求使用列表视图和详情视图实现，前者用于渲染搜索结果，后者用于生成各个项目的详情页面。

然而，最初的通用视图是函数，不能扩展。如果需求与列表视图相似但不完全一样，那么不得不自己从头实现。

Django 1.3 引入了基于类的视图，而且还通过基类、混入和拿来即用的具体类提供了一些通用视图类。这些基类和混入在 `django.views.generic` 包的 `base` 模块里，如图 12-4 所示。在这张图中，位于顶部的两个类，`View` 和 `TemplateResponseMixin`，负责完全不同的工作。

> 在 Classy Class-Based Views 网站（http://ccbv.co.uk）中可以深入研究这些类，你可以轻松地浏览各个视图类、查看它们的全部方法（继承的、覆盖的和自己添加的）、查看图表、浏览文档，以及跳转到 GitHub 中的源码（https://github.com/django/django/tree/master/django/views/generic）。

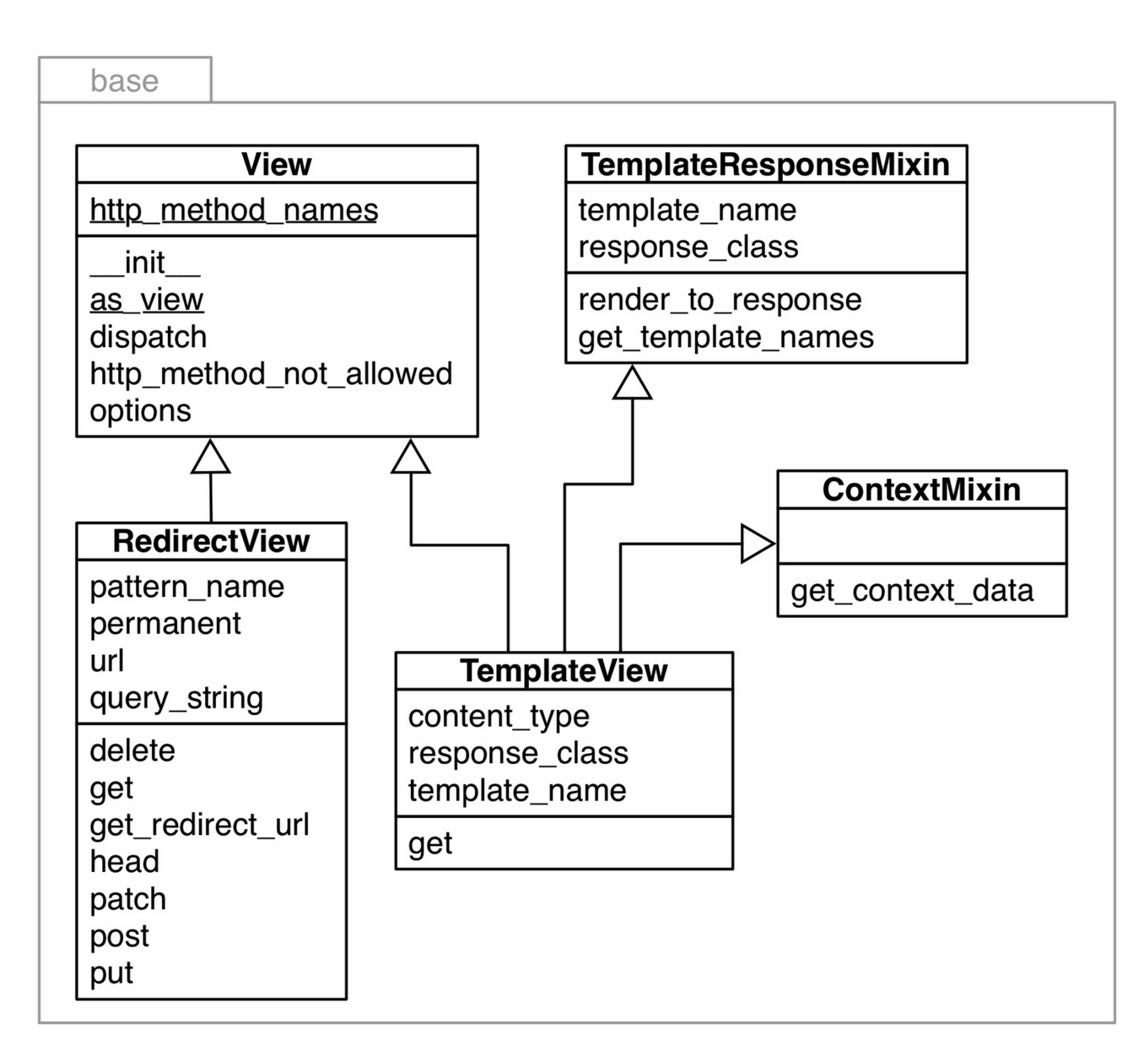


图 **12-4**：**`django.views.generic.base`** 模块的 **UML** 类图

`View` 是所有视图（可能是个抽象基类）的基类，提供核心功能，如 `dispatch` 方法。这个方法委托具体子类实现的处理方法（handler），如 `get`、`head`、`post` 等，处理不同的 HTTP 动词。[10]`RedirectView` 类只继承 `View`，可以看到，它实现了 `get`、`head`、`post` 等方法。

> [10]Django 程序员知道，`as_view` 类方法是 `View` 接口最为重要的部分，不过它与这里讨论的话题无关。

`View` 的具体子类应该实现处理方法，但它们为什么不在 `View` 接口中呢？原因是：子类只需实现它们想支持的处理方法。`TemplateView` 只

用于显示内容，因此它只实现了 get 方法。如果把 HTTP POST 请求发给 TemplateView，经继承的 View.dispatch 方法检查，它没有 post 处理方法，因此会返回 HTTP 405 Method Not Allowed（不允许使用的方法）响应。[11]

> [11]如果深入了解设计模式，你会发现 Django 的分派机制是动态版模板方法模式（https://en.wikipedia.org/wiki/Template_method_pattern）。之所以说是动态的，是因为 View 类不强制子类实现所有处理方法，而是让 dispatch 方法在运行时检查有没有针对特定请求的具体处理方法。

TemplateResponseMixin 提供的功能只针对需要使用模板的视图。例如，RedirectView 没有主体内容，因此它不需要模板，也就没有继承这个混入。TemplateResponseMixin 为 TemplateView 和 django.views.generic 包中定义的使用模板渲染的其他视图（例如 ListView、DetailView，等等）提供行为。图 12-5 是 django.views.generic.list 模块和部分 base 模块的图解。

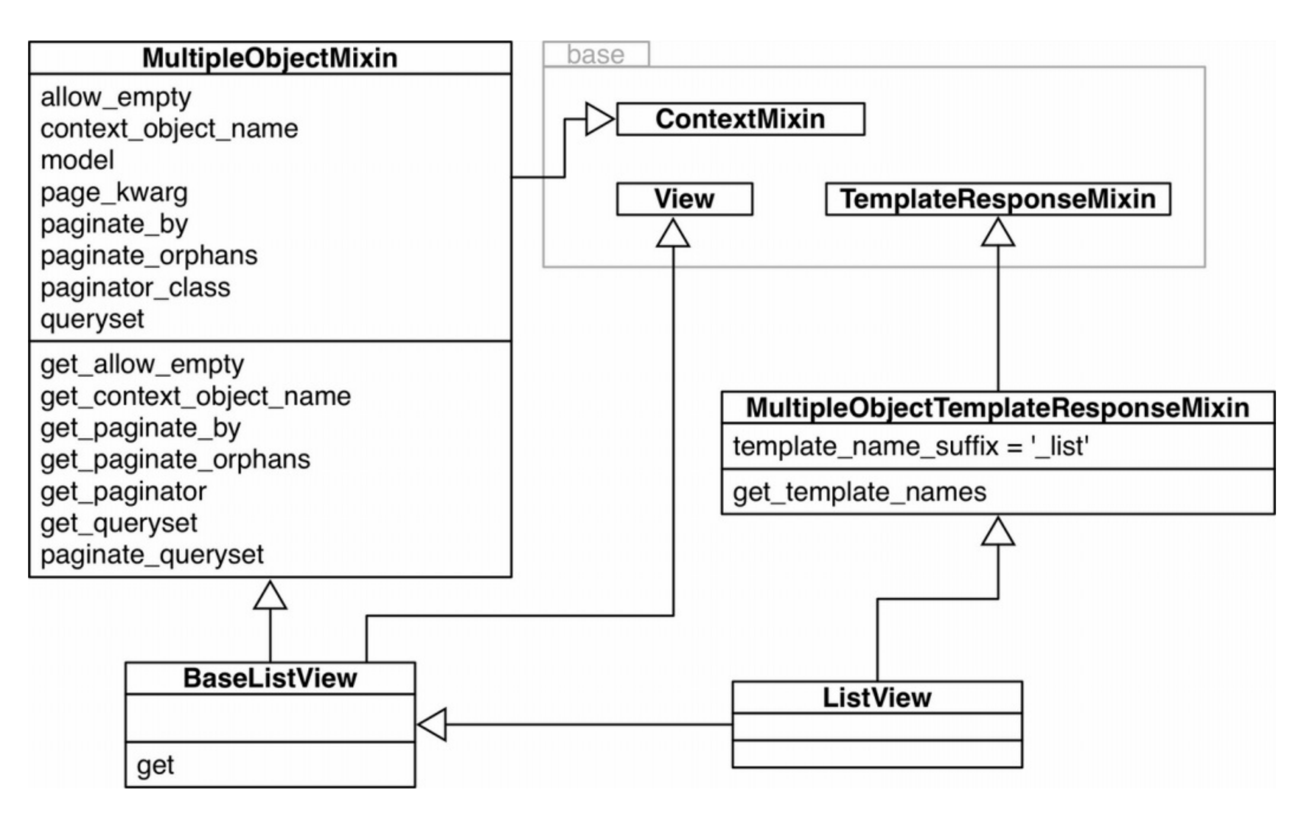


图 12-5：django.views.generic.list 模块的 UML 类图；图中属于 base 模块的三个类没有详细说明（参见图 12-4）；ListView 类没有方法和属性，它是一个聚合类

对 Django 用户来说，在图 12-5 中，最重要的类是 `ListView`。这是一个聚合类，不含任何代码（定义体中只有一个文档字符串）。`ListView` 实例有个 `object_list` 属性，模板会迭代它显示页面的内容，通常是数据库查询返回的多个对象。生成这个可迭代对象列表的相关功能都由 `MultipleObjectMixin` 提供。这个混入还提供了复杂的分页逻辑，即在一页中显示部分结果，并提供指向其他页面的链接。

假设你想创建一个使用模板渲染的视图，但是会生成一组 JSON 格式的对象，此时用得到 `BaseListView` 类。这个类提供了易于使用的扩展点，把 `View` 和 `MultipleObjectMixin` 的功能整合在一起，避免了模板机制的开销。

与 Tkinter 相比，Django 基于类的视图 API 是多重继承更好的示例。尤其是，Django 的混入类易于理解：各个混入的目的明确，而且名称的后缀都是 `...Mixin`。

Django 用户还没有完全拥抱基于类的视图。很多人确实在使用，但是用法有限，把它们当成黑盒；需要新功能时，很多 Django 程序员依然选择编写单块视图函数，负责处理所有事务，而不尝试重用基视图和混入。

学习基于类的视图和根据应用需求扩展它们确实需要一些时间，不过我觉得这是值得的：基于类的视图能避免大量样板代码，便于重用，还能增进团队交流——例如，为模板和传给模板上下文的变量定义标准的名称。基于类的视图把 Django 视图带到了正轨上。

我们对多重继承和混入类的讨论到此结束。

12.6　本章小结

本章对继承的讨论先从子类化内置类型引起的问题谈起：内置类型的原生方法使用 C 语言实现，不会调用子类中覆盖的方法，不过有极少数例外。因此，需要定制 `list`、`dict` 或 `str` 类型时，子类化 `UserList`、`UserDict` 或 `UserString` 更简单。这些类在 `collections` 模块（https://docs.python.org/3/library/collections.html）中定义，它们其实是对内置类型的包装，会把操作委托给内置类型——这是标准库中优先选择组合而不使用继承的三个例子。如果所需的行为与内置类型区别很大，或许更容易的做法是，子类化 `collections.abc` 模块（https://docs.python.org/3/library/collections.abc.html）中相应的抽象基类，然后自己实现。

本章余下的内容着重探讨了多重继承这把双刃剑。首先，我们说明了 `__mro__` 类属性中蕴藏的方法解析顺序，有了这一机制，继承方法的名称不再会发生冲突。我们还提到，内置的 `super()` 函数会按照 `__mro__` 属性给出的顺序调用超类的方法。然后，我们分析了 Python 标准库中 GUI 工具包 Tkinter 对多重继承的运用。Tkinter 不能代表当前的最佳实践，因此我们讨论了处理多重继承的一些方式，例如谨慎使用混入类，以及借助组合模式彻底避免使用多重继承。指出 Tkinter 对多重继承的使用已经到了滥用的程度后，我们在最后一节分析了 Django 基于类的视图，了解了它们的核心层次结构。我觉得这更好地利用了混入。

Lennart Regebro（一位经验非常丰富的 Python 程序员，也是本书的技术审校之一）发现 Django 通过混入设计的视图层次结构有点混乱。但是他又写道：

> 多重继承的危害和缺点被放大了。我从来不觉得它是什么大问题。

总之，每个人对如何使用以及要不要在自己的项目中使用多重继承都有自己的观点。但是，我们往往没得选择，因为我们必须使用的框架有它们自己的选择。

12.7　延伸阅读

使用抽象基类时，多重继承很常见，而且实际上也是不可避免的，因为最基本的集合抽象基类（Sequence、Mapping 和 Set）都扩展多个抽象基类。collections.abc 的源码（Lib/_collections_abc.py，https://hg.python.org/cpython/file/3.4/Lib/_collect
是抽象基类使用多重继承的范例——其中很多还是混入类。

Raymond Hettinger 写的文章“Python's super() considered super!”（https://rhettinger.wordpress.com/2011/05/26/super-considered-super/），从积极的角度解说了 Python 的 super 和多重继承的运作原理。这篇文章是对 James Knight 的“Python's Super is nifty, but you can't use it”（以前题为“Python's Super Considered Harmful”，https://fuhm.net/super-harmful/）一文作出的回应。

尽管这两篇文章的题目中提到了内置的 super 函数，但它不是真正的问题——Python 3 中的 super 函数没有 Python 2 中那么令人讨厌了。真正的问题是多重继承，它天生复杂，难以处理。Michele Simionato 不再批评这一点，他在“Setting Multiple Inheritance Straight”一文（http://www.artima.com/weblogs/viewpost.jsp?thread=246488）中给出了解决方案：他实现了性状（trait），这是一种受限的混入，源自 Self 语言。Simionato 写了一系列具有启发性的博客文章，对 Python 的多重继承进行了探讨，包括“The wonders of cooperative inheritance, or using super in Python 3”（http://www.artima.com/weblogs/viewpost.jsp?thread=281127），“Mixins considered harmful”第一部分（http://www.artima.com/weblogs/viewpost.jsp?thread=246341）和第二部分（http://www.artima.com/weblogs/viewpost.jsp?thread=246483），以及“Things to Know About Python Super”第一部分（http://www.artima.com/weblogs/viewpost.jsp?thread=236275）、第二部分（http://www.artima.com/weblogs/viewpost.jsp?thread=236278）和第三部分（http://www.artima.com/weblogs/viewpost.jsp?thread=237121）。最早的文章使用 Python 2 的 super 句法，不过依然值得一读。

我读过 Grady Booch 写的《面向对象分析与设计（第 3 版）》，强烈推荐给你，这是面向对象思维的通用入门书，与具体的编程语言无关。很

少有书能这样不带偏见地讨论多重继承。

杂谈

想想哪些类是真正需要的

大多数程序员编写应用程序而不开发框架。即便是开发框架的那些人，多数时候（或大多数时候）也是在编写应用程序。编写应用程序时，我们通常不用设计类的层次结构。我们至多会编写子类、继承抽象基类或框架提供的其他类。作为应用程序开发者，我们极少需要编写作为其他类的超类的类。我们自己编写的类几乎都是末端类（即继承树的叶子）。

如果作为应用程序开发者，你发现自己在构建多层类层次结构，可能是发生了下述事件中的一个或多个。

- 你在重新发明轮子。去找框架或库，它们提供的组件可以在应用程序中重用。
- 你使用的框架设计不良。去寻找替代品。
- 你在过度设计。记住要遵守 **KISS** 原则。
- 你厌烦了编写应用程序，决定新造一个框架。恭喜，祝你好运！

这些事情你可能都会遇到：你厌倦了，决定重新发明轮子，自己构建设计过度和不良的框架，因此不得不编写一个又一个类去解决鸡毛蒜皮的小事。希望你能乐在其中，至少得到应有的回报。

内置类型的不当行为是缺陷还是特性

内置的 `dict`、`list` 和 `str` 类型是 Python 的底层基础，因此速度必须快，与这些内置类型有关的任何性能问题几乎都会对其他所有代码产生重大影响。于是，CPython 走了捷径，故意让内置类型的方法行为不当，即不调用被子类覆盖的方法。解决这一困境的可能方式之一是，为这些类型分别提供两种实现：一种供内部使用，为解释器做了优化；另一种供外部使用，便于扩展。

但是等等，我们已经拥有这些了：`UserDict`、`UserList` 和 `UserString` 虽然没有内置类型的速度快，但是易于扩展。CPython 采用的这种务实方式意味着，我们也要在自己的应用程序中使用做了优化但是难以子类化的实现。这是合理的，因为我们每天都使用 `dict`、`list` 和 `str`，但是很少需要定制映射、列表或字符串。我们只需知道其中涉及的取舍。

其他语言对继承的支持

“面向对象”这个术语是 Alan Kay 发明的，而 Smalltalk 只支持单继承，不过有些派生版以不同的方式支持多重继承，例如现代的 Squeak 和 Smalltalk 方言 Pharo 支持性状（trait）——这是实现混入类的语言结构，而且能避免多重继承的一些问题。

C++ 是第一门实现多重继承的流行语言，但是这一功能被滥用了，因此意欲取代 C++ 的 Java 不支持多重继承（即没有混入类）。不过，Java 8 引入了默认方法，这使得接口与 C++ 和 Python 用于定义接口的抽象类十分相似。但是它们之间有个关键的区别：Java 的接口没有状态。Java 之后，使用最广泛的 JVM 语言要数 Scala 了，而它实现了性状。支持性状的其他语言还有最新稳定版 PHP 和 Groovy，以及正在开发的 Rust 和 Perl 6。因此可以说，性状是目前的趋势。

Ruby 对多重继承的态度很明确：对其不支持，但是引入了混入。Ruby 类的定义体中可以包含模块，这样模块中定义的方法就变成了类实现的一部分。这是“纯粹”的混入，不涉及继承，因此 Ruby 混入显然不会影响所在类的类型。这种方式凸显了混入的优点，避免了很多常见问题。

最近广受瞩目的两门语言——Go 和 Julia——对继承的支持极其有限。Go 完全不支持继承，但是它实现的接口与静态鸭子类型相似（详情参见第 11 章的“杂谈”）。Julia 回避“类”（class）这个术语，只接受“类型”（type）。Julia 有类型层次结构，但是子类型不能继承结构，只能继承行为，而且只能为抽象类型创建子类型。此外，Julia 的方法使用多重分派，这是 7.8.2 节所述机制的高级形式。

第 13 章　正确重载运算符

> 有些事情让我不安，比如运算符重载。我决定不支持运算符重载，这完全是个人选择，因为我见过太多 C++ 程序员滥用它。[1]
>
> ——James Gosling
> Java 之父

[1]摘自“The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling”一文（http://www.gotw.ca/publications/c_family_interview.htm）。

运算符重载的作用是让用户定义的对象使用中缀运算符（如 `+` 和 `|`）或一元运算符（如 `-` 和 `~`）。说得宽泛一些，在 Python 中，函数调用（`()`）、属性访问（`.`）和元素访问 / 切片（`[]`）也是运算符，不过本章只讨论一元运算符和中缀运算符。

在 1.2.1 节，我们为 `Vector` 类简略实现了几个运算符。示例 1-2 中的 `__add__` 和 `__mul__` 方法是为了展示如何使用特殊方法重载运算符，不过有些小问题被我们忽视了。此外，在示例 9-2 中，我们定义的 `Vector2d.__eq__` 方法认为 `Vector(3, 4) == [3, 4]` 是真的（`True`），这可能并不合理。本章会解决这些问题。

在接下来的几节，我们将讨论：

- Python 如何处理中缀运算符中不同类型的操作数
- 使用鸭子类型或显式类型检查处理不同类型的操作数
- 中缀运算符如何表明自己无法处理操作数
- 众多比较运算符（如 `==`、`>`、`<=`，等等）的特殊行为
- 增量赋值运算符（如 `+=`）的默认处理方式和重载方式

13.1　运算符重载基础

在某些圈子中，运算符重载的名声并不好。这个语言特性可能（已经）被滥用，让程序员困惑，导致缺陷和意料之外的性能瓶颈。但是，如果使用得当，API 会变得好用，代码会变得易于阅读。Python 施加了一些限制，做好了灵活性、可用性和安全性方面的平衡：

- 不能重载内置类型的运算符
- 不能新建运算符，只能重载现有的
- 某些运算符不能重载——`is`、`and`、`or` 和 `not`（不过位运算符 `&`、`|` 和 `~` 可以）

第 10 章已经为 `Vector` 定义了一个中缀运算符，即 `==`，这个运算符由 `__eq__` 方法支持。本章将改进 `__eq__` 方法的实现，更好地处理不是 `Vector` 实例的操作数。然而，在运算符重载方面，众多比较运算符（`==`、`!=`、`>`、`<`、`>=`、`<=`）是特例，因此我们首先将在 `Vector` 中重载四个算术运算符：一元运算符 `-` 和 `+`，以及中缀运算符 `+` 和 `*`。

先从最简单的入手：一元运算符。

13.2　一元运算符

在 Python 语言参考手册中，“6.5. Unary arithmetic and bitwise operations”一节[2]（https://docs.python.org/3/reference/expressions.html#unary-arithmetic-and-bitwise-operations）列出了三个一元运算符。下面是这三个运算符和对应的特殊方法。

> [2]现有版本是 6.6 节，而不是 6.5 节。——编者注

`-`（`__neg__`）

　　一元取负算术运算符。如果 `x` 是 `-2`，那么 `-x == 2`。

`+`（`__pos__`）

　　一元取正算术运算符。通常，`x == +x`，但也有一些例外。如果好奇，请阅读“`x` 和 `+x` 何时不相等”附注栏。

`~`（`__invert__`）

　　对整数按位取反，定义为 `~x == -(x+1)`。如果 `x` 是 `2`，那么 `~x == -3`。

Python 语言参考手册中的“Data Model”一章（https://docs.python.org/3/reference/datamodel.html#object.__neg__）还把内置的 `abs(...)` 函数列为一元运算符。它对应的特殊方法是 `__abs__`，从 1.2.1 节起已经见过多次。

支持一元运算符很简单，只需实现相应的特殊方法。这些特殊方法只有一个参数，`self`。然后，使用符合所在类的逻辑实现。不过，要遵守运算符的一个基本规则：始终返回一个新对象。也就是说，不能修改 `self`，要创建并返回合适类型的新实例。

对 `-` 和 `+` 来说，结果可能是与 `self` 同属一类的实例。多数时候，`+` 最

好返回 self 的副本。abs(...) 的结果应该是一个标量。但是对 ~ 来说，很难说什么结果是合理的，因为可能不是处理整数的位，例如在 ORM 中，SQL WHERE 子句应该返回反集。

如前所述，我们将为第 10 章定义的 Vector 类实现几个新运算符。示例 13-1 列出了示例 10-16 实现的 __abs__ 方法，以及新增加的 __neg__ 和 __pos__ 一元运算符方法。

示例 **13-1**　vector_v6.py：把一元运算符 - 和 + 添加到示例 10-16 中

```
    def __abs__(self):
        return math.sqrt(sum(x * x for x in self))

    def __neg__(self):
        return Vector(-x for x in self)  ❶

    def __pos__(self):
        return Vector(self)  ❷
```

❶ 为了计算 -v，构建一个新 Vector 实例，把 self 的每个分量都取反。

❷ 为了计算 +v，构建一个新 Vector 实例，传入 self 的各个分量。

还记得吗？ Vector 实例是可迭代的对象，而且 Vector.__init__ 的参数是一个可迭代对象，因此 __neg__ 和 __pos__ 的实现短小精悍。

我们不打算实现 __invert__ 方法，因此如果用户在 Vector 实例上尝试计算 ~v，Python 会抛出 TypeError，而且输出明确的错误消息，“bad operand type for unary ~: 'Vector'”。

下述附注栏讨论一个奇怪的问题，能增长你的 + 一元运算符知识。接下来的重要话题是：重载向量加法运算符 +（见 13.3 节）。

x 和 +x 何时不相等

每个人都觉得 x == +x，而且在 Python 中，几乎所有情况下都是

这样。但是，我在标准库中找到两例 `x != +x` 的情况。

第一例与 `decimal.Decimal` 类有关。如果 `x` 是 `Decimal` 实例，在算术运算的上下文中创建，然后在不同的上下文中计算 `+x`，那么 `x != +x`。例如，`x` 所在的上下文使用某个精度，而计算 `+x` 时，精度变了，如示例 13-2 所示。

示例 **13-2** 算术运算上下文的精度变化可能导致 `x` 不等于 `+x`

```
>>> import decimal
>>> ctx = decimal.getcontext()  ❶
>>> ctx.prec = 40  ❷
>>> one_third = decimal.Decimal('1') / decimal.Decimal('3')  ❸
>>> one_third  ❹
Decimal('0.3333333333333333333333333333333333333333')
>>> one_third == +one_third  ❺
True
>>> ctx.prec = 28  ❻
>>> one_third == +one_third  ❼
False
>>> +one_third  ❽
Decimal('0.3333333333333333333333333333')
```

❶ 获取当前全局算术运算的上下文引用。

❷ 把算术运算上下文的精度设为 40。

❸ 使用当前精度计算 1/3。

❹ 查看结果，小数点后有 40 个数字。

❺ `one_third == +one_third` 返回 `True`。

❻ 把精度降低为 28，这是 Python 3.4 为 `Decimal` 算术运算设定的默认精度。

❼ 现在，`one_third == +one_third` 返回 `False`。

❽ 查看 `+one_third`，小数点后有 28 个数字。

虽然每个 `+one_third` 表达式都会使用 `one_third` 的值创建一个

新 Decimal 实例，但是会使用当前算术运算上下文的精度。

x != +x 的第二例在 collections.Counter 的文档中（https://docs.python.org/3/library/collections.html#collections.Counter）。类实现了几个算术运算符，例如中缀运算符 +，作用是把两个 Counter 实例的计数器加在一起。然而，从实用角度出发，Counter 相加时，负值和零值计数会从结果中剔除。而一元运算符 + 等同于加上一个空 Counter，因此它产生一个新的 Counter 且仅保留大于零的计数器。见示例 13-3。

示例 **13-3**　一元运算符 + 得到一个新 Counter 实例，但是没有零值和负值计数器[3]

```
>>> ct = Counter('abracadabra')
>>> ct
Counter({'a': 5, 'r': 2, 'b': 2, 'd': 1, 'c': 1})
>>> ct['r'] = -3
>>> ct['d'] = 0
>>> ct
Counter({'a': 5, 'b': 2, 'c': 1, 'd': 0, 'r': -3})
>>> +ct
Counter({'a': 5, 'b': 2, 'c': 1})
```

下面回归正题。

[3]应该在最前面加一行：>>> from collections import Counter。——编者注

13.3　重载向量加法运算符+

Vector 类是序列类型，按照“Data Model”一章中的“3.3.6. Emulating container types”一节（https://docs.python.org/3/reference/datamodel.html#emulating-container-types）所说，序列应该支持 + 运算符（用于拼接），以及 * 运算符（用于重复复制）。然而，我们将使用向量数学运算实现 + 和 * 运算符。这么做更难一些，但是对 Vector 类型来说更有意义。

两个欧几里得向量加在一起得到的是一个新向量，它的各个分量是两个向量中相应的分量之和。比如说：

```
>>> v1 = Vector([3, 4, 5])
>>> v2 = Vector([6, 7, 8])
>>> v1 + v2
Vector([9.0, 11.0, 13.0])
>>> v1 + v2 == Vector([3+6, 4+7, 5+8])
True
```

如果尝试把两个不同长度的 Vector 实例加在一起会怎样？此时可以抛出错误，但是根据实际运用情况（例如信息检索），最好使用零填充较短的那个向量。我们想要的效果是这样：

```
>>> v1 = Vector([3, 4, 5, 6])
>>> v3 = Vector([1, 2])
>>> v1 + v3
Vector([4.0, 6.0, 5.0, 6.0])
```

确定这些基本的要求之后，__add__ 方法的实现短小精悍，如示例 13-4 所示。

示例 **13-4**　Vector.__add__ 方法，第 1 版

```
# 在Vector类中定义

def __add__(self, other):
    pairs = itertools.zip_longest(self, other, fillvalue=0.0) #  ❶
    return Vector(a + b for a, b in pairs) #  ❷
```

❶ pairs 是个生成器，它会生成 (a, b) 形式的元组，其中 a 来自 self，b 来自 other。如果 self 和 other 的长度不同，使用 fillvalue 填充较短的那个可迭代对象。

❷ 构建一个新 Vector 实例，使用生成器表达式计算 pairs 中各个元素的和。

注意，__add__ 返回一个新 Vector 实例，而没有影响 self 或 other。

> 实现一元运算符和中缀运算符的特殊方法一定不能修改操作数。使用这些运算符的表达式期待结果是新对象。只有增量赋值表达式可能会修改第一个操作数（self），参见 13.6 节。

示例 13-4 中的实现方式可以把 Vector 加到 Vector2d 上，还可以把 Vector 加到元组或任何生成数字的可迭代对象上，如示例 13-5 所示。

示例 **13-5**　第 1 版 Vector.__add__ 方法也支持 Vector 之外的对象

```
>>> v1 = Vector([3, 4, 5])
>>> v1 + (10, 20, 30)
Vector([13.0, 24.0, 35.0])
>>> from vector2d_v3 import Vector2d
>>> v2d = Vector2d(1, 2)
>>> v1 + v2d
Vector([4.0, 6.0, 5.0])
```

示例 13-5 中的两个加法都能如我们所期待的那样计算，这是因为 __add__ 使用了 zip_longest(...)，它能处理任何可迭代对象，而且构建新 Vector 实例的生成器表达式仅仅是把 zip_longest(...) 生成

的值对相加（a + b），因此可以使用任何生成数字元素的可迭代对象。

然而，如果对调操作数（见示例 13-6），混合类型的加法就会失败。

示例 **13-6**　如果左操作数是 Vector 之外的对象，第一版 Vector.__add__ 方法无法处理

```
>>> v1 = Vector([3, 4, 5])
>>> (10, 20, 30) + v1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "Vector") to tuple
>>> from vector2d_v3 import Vector2d
>>> v2d = Vector2d(1, 2)
>>> v2d + v1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Vector2d' and 'Vector'
```

为了支持涉及不同类型的运算，Python 为中缀运算符特殊方法提供了特殊的分派机制。对表达式 a + b 来说，解释器会执行以下几步操作（见图 13-1）。

(1) 如果 a 有 __add__ 方法，而且返回值不是 NotImplemented，调用 a.__add__(b)，然后返回结果。

(2) 如果 a 没有 __add__ 方法，或者调用 __add__ 方法返回 NotImplemented，检查 b 有没有 __radd__ 方法，如果有，而且没有返回 NotImplemented，调用 b.__radd__(a)，然后返回结果。

(3) 如果 b 没有 __radd__ 方法，或者调用 __radd__ 方法返回 NotImplemented，抛出 TypeError，并在错误消息中指明操作数类型不支持。

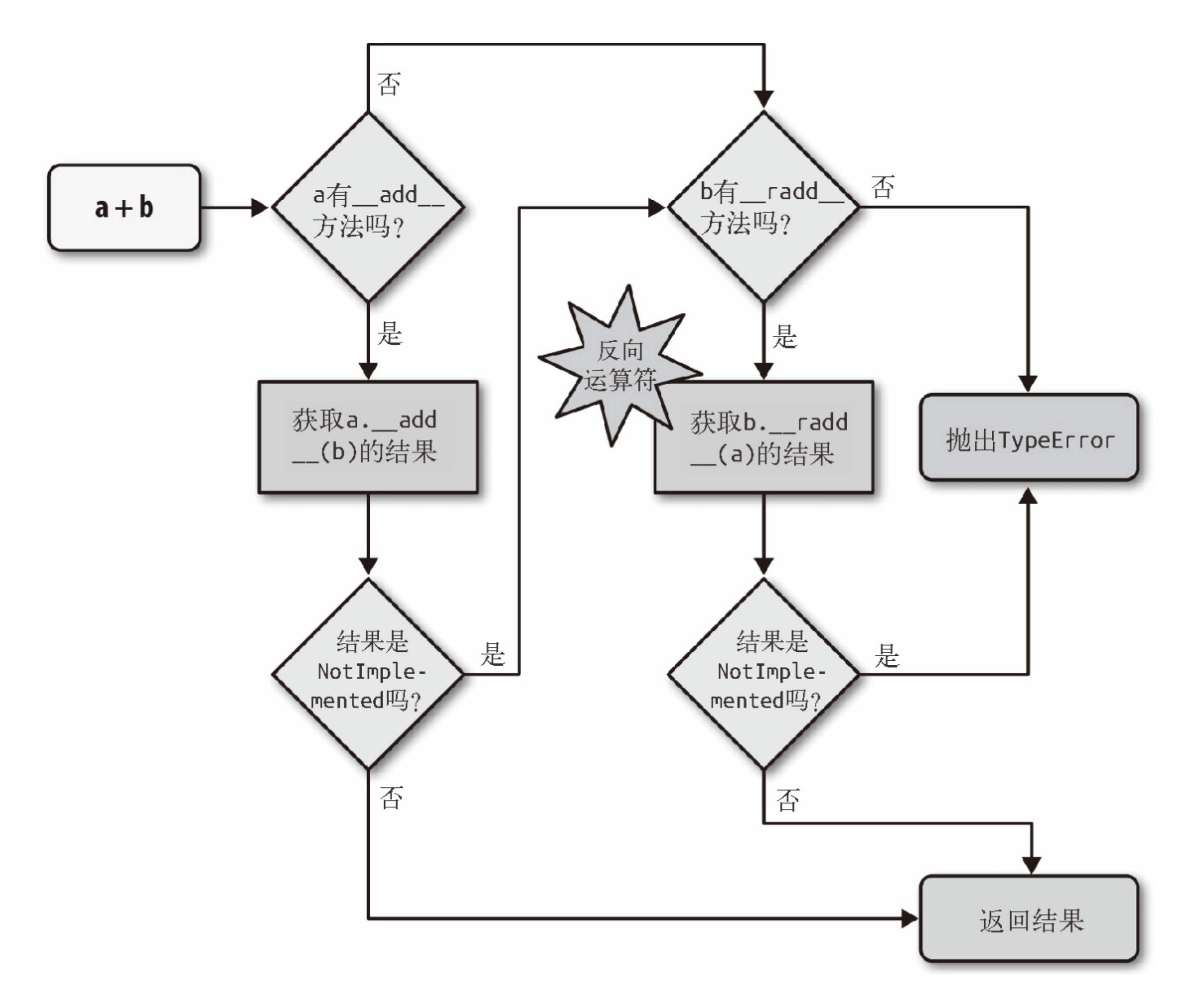


图 13-1：使用 `__add__` 和 `__radd__` 计算 `a + b` 的流程图

`__radd__` 是 `__add__` 的“反射”（reflected）版本或“反向”（reversed）版本。我喜欢把它叫作“反向”特殊方法。[4] 本书的三位技术审校，Alex、Anna 和 Leo 告诉我，他们喜欢称之为“右向”（right）特殊方法，因为他们在右操作数上调用。不管你喜欢哪个以“r”开头的单词，`__radd__` 和 `__rsub__` 等类似方法中的“r”就是这个意思。

> [4]这两个术语在 Python 文档中都使用过。“Data Model”一章（https://docs.python.org/3/reference/datamodel.html）用的是“reflected”（反射），而 `numbers` 模块文档的“9.1.2.2. Implementing the arithmetic operations”一节（https://docs.python.org/3/library/numbers.html#implementing-the-arithmetic-operations）用的是“forward”（正向）方法和“reverse”（反向）方法。我觉得后者更好，因为“正向”和“反向”明确指出了方向，而“反射”就没这种效果。

因此，为了让示例 13-6 中的混合类型加法能正确计算，我们要实现 `Vector.__radd__` 方法。这是一种后备机制，如果左操作数没有实现 `__add__` 方法，或者实现了，但是返回 `NotImplemented` 表明它不知道如何处理右操作数，那么 Python 会调用 `__radd__` 方法。

> 别把 `NotImplemented` 和 `NotImplementedError` 搞混了。前者是特殊的单例值，如果中缀运算符特殊方法不能处理给定的操作数，那么要把它返回（`return`）给解释器。而 `NotImplementedError` 是一种异常，抽象类中的占位方法把它抛出（`raise`），提醒子类必须覆盖。

最简可用的 `__radd__` 实现如示例 13-7 所示。

示例 **13-7**　`Vector.__add__` 和 `__radd__` 方法

```
# 在Vector类中定义

def __add__(self, other): #  ❶
    pairs = itertools.zip_longest(self, other, fillvalue=0.0)
    return Vector(a + b for a, b in pairs)

def __radd__(self, other):  # ❷
    return self + other
```

❶ `__add__` 方法与示例 13-4 中一样，没有变化；这里列出，是因为 `__radd__` 要用它。

❷ `__radd__` 直接委托 `__add__`。

`__radd__` 通常就这么简单：直接调用适当的运算符，在这里就是委托 `__add__`。任何可交换的运算符都能这么做。处理数字和向量时，`+` 可以交换，但是拼接序列时不行。

示例 13-4 中的方法可以处理 `Vector` 对象或任何具有数值元素的可迭代对象，例如 `Vector2d` 实例、整数元组或浮点数数组。但是，如果提供的对象不可迭代，那么 `__add__` 就无法处理，而且提供的错误消息不是很有用，如示例 13-8 所示。

示例 **13-8** `Vector.__add__` 方法的操作数要是可迭代对象

```
>>> v1 + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "vector_v6.py", line 328, in __add__
    pairs = itertools.zip_longest(self, other, fillvalue=0.0)
TypeError: zip_longest argument #2 must support iteration
```

如果一个操作数是可迭代对象，但是它的元素不能与 `Vector` 中的浮点数元素相加，给出的消息也没什么用。如示例 13-9 所示。

示例 **13-9** `Vector.__add__` 方法的操作数应是可迭代的数值对象

```
>>> v1 + 'ABC'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "vector_v6.py", line 329, in __add__
    return Vector(a + b for a, b in pairs)
  File "vector_v6.py", line 243, in __init__
    self._components = array(self.typecode, components)
  File "vector_v6.py", line 329, in <genexpr>
    return Vector(a + b for a, b in pairs)
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

示例 13-8 和示例 13-9 揭露的问题比晦涩难懂的错误消息严重：如果由于类型不兼容而导致运算符特殊方法无法返回有效的结果，那么应该返回 `NotImplemented`，而不是抛出 `TypeError`。返回 `NotImplemented` 时，另一个操作数所属的类型还有机会执行运算，即 Python 会尝试调用反向方法。

为了遵守鸭子类型精神，我们不能测试 `other` 操作数的类型，或者它的元素的类型。我们要捕获异常，然后返回 `NotImplemented`。如果解释器还未反转操作数，那么它将尝试去做。如果反向方法返回 `NotImplemented`，那么 Python 会抛出 `TypeError`，并返回一个标准的错误消息，例如“`unsupported operand type(s) for +: Vector and str`”。

示例 13-10 是实现 `Vector` 加法的特殊方法的最终版。

示例 **13-10**　vector_v6.py：**+** 运算符方法，添加到 vector_v5.py（见示例 10-16）中

```
    def __add__(self, other):
        try:
            pairs = itertools.zip_longest(self, other, fillvalue=0.0)
            return Vector(a + b for a, b in pairs)
        except TypeError:
            return NotImplemented

    def __radd__(self, other):
        return self + other
```

如果中缀运算符方法抛出异常，就终止了运算符分派机制。对 `TypeError` 来说，通常最好将其捕获，然后返回 `NotImplemented`。这样，解释器会尝试调用反向运算符方法，如果操作数是不同的类型，对调之后，反向运算符方法可能会正确计算。

至此，我们编写了 `__add__` 和 `__radd__` 方法，安全重载了 **+** 运算符。接下来实现另一个中缀运算符：`*`。

13.4 重载标量乘法运算符*

Vector([1, 2, 3]) * x 是什么意思？如果 x 是数字，就是计算标量积（scalar product），结果是一个新 Vector 实例，各个分量都会乘以 x——这也叫元素级乘法（elementwise multiplication）。

```
>>> v1 = Vector([1, 2, 3])
>>> v1 * 10
Vector([10.0, 20.0, 30.0])
>>> 11 * v1
Vector([11.0, 22.0, 33.0])
```

涉及 Vector 操作数的积还有一种，叫两个向量的点积（dot product）；如果把一个向量看作 $1 \times N$ 矩阵，把另一个向量看作 $N \times 1$ 矩阵，那么就是矩阵乘法。NumPy 等库目前的做法是，不重载这两种意义的 *，只用 * 计算标量积。例如，在 NumPy 中，点积使用 numpy.dot() 函数计算。[5]

> [5]从 Python 3.5 起，@ 记号可以用作中缀点积运算符。详情参见“Python 3.5 新引入的中缀运算符 @”附注栏。

回到标量积的话题。我们依然先实现最简可用的 __mul__ 和 __rmul__ 方法：

```
# 在Vector类中定义

def __mul__(self, scalar):
    return Vector(n * scalar for n in self)

def __rmul__(self, scalar):
    return self * scalar
```

这两个方法确实可用，但是提供不兼容的操作数时会出问题。scalar 参数的值要是数字，与浮点数相乘得到的积是另一个浮点数（因为 Vector 类在内部使用浮点数数组）。因此，不能使用复数，但可以是

int、bool（int 的子类），甚至 fractions.Fraction 实例等标量。

我们可以像示例 13-10 那样，采用鸭子类型技术，在 __mul__ 方法中捕获 TypeError。但是，这个问题有个更易于理解的方式，而且也更合理：白鹅类型。我们将使用 isinstance() 检查 scalar 的类型，但是不硬编码具体的类型，而是检查 numbers.Real 抽象基类。这个抽象基类涵盖了我们所需的全部类型，而且还支持以后声明为 numbers.Real 抽象基类的真实子类或虚拟子类的数值类型。示例 13-11 展示了白鹅类型的实际运用——显式检查抽象类型。完整的代码清单参见本书的代码仓库。

你可能还记得 11.6 节说过，decimal.Decimal 没有把自己注册为 numbers.Real 的虚拟子类。因此，Vector 类不会处理 decimal.Decimal 数字。

示例 **13-11**　vector_v7.py：增加 * 运算符方法

```
from array import array
import reprlib
import math
import functools
import operator
import itertools
import numbers  # ❶

class Vector:
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components)

    # 排版需要，省略了很多方法
    # 参见https://github.com/fluentpython/example-code中的vector_v7.py

    def __mul__(self, scalar):
        if isinstance(scalar, numbers.Real):  # ❷
            return Vector(n * scalar for n in self)
        else:  # ❸
            return NotImplemented
```

```
    def __rmul__(self, scalar):
        return self * scalar  # ❹
```

❶ 为了检查类型，导入 `numbers` 模块。

❷ 如果 `scalar` 是 `numbers.Real` 某个子类的实例，用分量的乘积创建一个新 `Vector` 实例。

❸ 否则，返回 `NotImplemented`，让 Python 尝试在 `scalar` 操作数上调用 `__rmul__` 方法。

❹ 这里，`__rmul__` 方法只需执行 `self * scalar`，委托给 `__mul__` 方法。

有了示例 13-11 中的代码之后，我们可以拿 `Vector` 实例乘以常规的标量值和不那么寻常的数字类型了：

```
>>> v1 = Vector([1.0, 2.0, 3.0])
>>> 14 * v1
Vector([14.0, 28.0, 42.0])
>>> v1 * True
Vector([1.0, 2.0, 3.0])
>>> from fractions import Fraction
>>> v1 * Fraction(1, 3)
Vector([0.3333333333333333, 0.6666666666666666, 1.0])
```

通过实现 `+` 和 `*`，我们讲解了编写中缀运算符最常用的模式。`+` 和 `*` 用的技术对表 13-1 中列出的所有运算符都适用（就地运算符在 13.6 节讨论）。

表**13-1**：中缀运算符方法的名称（就地运算符用于增量赋值；比较运算符在表**13-2**中）

运算符	正向方法	反向方法	就地方法	说明
`+`	`__add__`	`__radd__`	`__iadd__`	加法或拼接

-	__sub__	__rsub__	__isub__	减法
*	__mul__	__rmul__	__imul__	乘法或重复复制
/	__truediv__	__rtruediv__	__itruediv__	除法
//	__floordiv__	__rfloordiv__	__ifloordiv__	整除
%	__mod__	__rmod__	__imod__	取模
divmod()	__divmod__	__rdivmod__	__idivmod__	返回由整除的商和模数组成的元组
**, pow()	__pow__	__rpow__	__ipow__	取幂*
@	__matmul__	__rmatmul__	__imatmul__	矩阵乘法#
&	__and__	__rand__	__iand__	位与
|	__or__	__ror__	__ior__	位或
^	__xor__	__rxor__	__ixor__	位异或
<<	__lshift__	__rlshift__	__ilshift__	按位左移
>>	__rshift__	__rrshift__	__irshift__	按位右移

* pow 的第三个参数 modulo 是可选的：pow(a, b, modulo)，直接调用特殊方法时也支持这个参数（如 a.__pow__(b, modulo)）。

Python 3.5 新引入的。

众多比较运算符也是一类中缀运算符，但是规则稍有不同。我们将在下一节讨论众多比较运算符。

下述附注栏介绍了 Python 3.5（写作本书时尚未发布 [6]）引入的 @ 运算符，选读。

> [6]现已发布。——编者注

Python 3.5 新引入的中缀运算符 **@**

Python 3.4 没有为点积提供中缀运算符。不过，写作本书时，Python 3.5 的 pre-alpha 版实现了“PEP 465 — A dedicated infix operator for matrix multiplication”（https://www.python.org/dev/peps/pep-0465/），提供了点积所需的 @ 记号（例如，`a @ b` 是 a 和 b 的点积）。@ 运算符由特殊方法 `__matmul__`、`__rmatmul__` 和 `__imatmul__` 提供支持，名称取自“matrix multiplication”（矩阵乘法）。目前，标准库还没用到这些方法，但是 Python 3.5 的解释器能识别，因此 NumPy 团队（以及我们自己）可以在用户定义的类型中支持 @ 运算符。Python 解析器也做了修改，能处理中缀运算符 @（在 Python 3.4 中，`a @ b` 是一种句法错误）。

为了体验一下，我从源码编译了 Python 3.5，然后为 `Vector` 实现了点积运算符 @，还做了测试。

下面是我做的最简单的测试：

```
>>> va = Vector([1, 2, 3])
>>> vz = Vector([5, 6, 7])
>>> va @ vz == 38.0  # 1*5 + 2*6 + 3*7
True
>>> [10, 20, 30] @ vz
380.0
>>> va @ 3
Traceback (most recent call last):
  ...
TypeError: unsupported operand type(s) for @: 'Vector' and 'int'
```

下面是相应特殊方法的代码：

```
class Vector:
    # 排版需要，省略了很多方法

    def __matmul__(self, other):
        try:
            return sum(a * b for a, b in zip(self, other))
        except TypeError:
            return NotImplemented

    def __rmatmul__(self, other):
        return self @ other
```

完整的源码在本书代码仓库（https://github.com/fluentpython/example-code）里的 vector_py3_5.py 文件中。

记得要在 Python 3.5 中测试，否则会导致 `SyntaxError`！

13.5　众多比较运算符

Python 解释器对众多比较运算符（`==`、`!=`、`>`、`<`、`>=`、`<=`）的处理与前文类似，不过在两个方面有重大区别。

- 正向和反向调用使用的是同一系列方法。这方面的规则如表 13-2 所示。例如，对 `==` 来说，正向和反向调用都是 `__eq__` 方法，只是把参数对调了；而正向的 `__gt__` 方法调用的是反向的 `__lt__` 方法，并把参数对调。

- 对 `==` 和 `!=` 来说，如果反向调用失败，Python 会比较对象的 ID，而不抛出 `TypeError`。

表**13-2**：众多比较运算符：正向方法返回**`NotImplemented`**的话，调用反向方法

分组	中缀运算符	正向方法调用	反向方法调用	后备机制
相等性	`a == b`	`a.__eq__(b)`	`b.__eq__(a)`	返回 `id(a) == id(b)`
	`a != b`	`a.__ne__(b)`	`b.__ne__(a)`	返回 `not (a == b)`
排序	`a > b`	`a.__gt__(b)`	`b.__lt__(a)`	抛出 `TypeError`
	`a < b`	`a.__lt__(b)`	`b.__gt__(a)`	抛出 `TypeError`
	`a >= b`	`a.__ge__(b)`	`b.__le__(a)`	抛出 `TypeError`
	`a <= b`	`a.__le__(b)`	`b.__ge__(a)`	抛出 `TypeError`



Python 3 的新行为

Python 2 之后的比较运算符后备机制都变了。对于 `__ne__`，现在 Python 3 返回结果是对 `__eq__` 结果的取反。对于排序比较运算符，Python 3 抛出 `TypeError`，并把错误消息设为 `'unorderable types: int() < tuple()'`。在 Python 2 中，这些比较的结果很怪异，会考虑对象的类型和 ID，而且无规律可循。然而，比较整数和元组确实没有意义，因此此时抛出 `TypeError` 是这门语言的一大进步。

了解这些规则之后，我们来分析并改进 `Vector.__eq__` 方法的行为。这个方法在 vector_v5.py 中是这样定义的：

```
class Vector:
    # 省略了很多行

    def __eq__(self, other):
        return (len(self) == len(other) and
                all(a == b for a, b in zip(self, other)))
```

这个方法的行为如示例 13-12 所示。

示例 **13-12**　`Vector` 实例与 `Vector` 实例、`Vector2d` 实例和元组比较

```
>>> va = Vector([1.0, 2.0, 3.0])
>>> vb = Vector(range(1, 4))
>>> va == vb  # ❶
True
>>> vc = Vector([1, 2])
>>> from vector2d_v3 import Vector2d
>>> v2d = Vector2d(1, 2)
>>> vc == v2d  # ❷
True
>>> t3 = (1, 2, 3)
>>> va == t3  # ❸
True
```

❶ 两个具有相同数值分量的 `Vector` 实例是相等的。

❷ 如果 Vector 实例的分量与 Vector2d 实例的分量都相等，那么两个实例相等。[7]

> [7]实际运行时会抛出异常：TypeError: object of type 'Vector2d' has no len()，因为 Vector2d 没有实现 __len__ 特殊方法。如果改为 vc == set(v2d) 就会返回 True。——编者注

❸ Vector 实例的分量与元组或其他任何可迭代对象的元素相等，那么对象也相等。

示例 13-12 中的最后一个结果可能不是很理想。我对这一点没有强制规则，要由应用上下文决定。不过，“Python 之禅”说道：

> 如果存在多种可能，不要猜测。

对操作数过度宽容可能导致令人惊讶的结果，而程序员讨厌惊喜。

从 Python 自身来找线索，我们发现 [1,2] == (1, 2) 的结果是 False。因此，我们要保守一点，做些类型检查。如果第二个操作数是 Vector 实例（或者 Vector 子类的实例），那么就使用 __eq__ 方法的当前逻辑。否则，返回 NotImplemented，让 Python 处理。参见示例 13-13。

示例 **13-13**　vector_v8.py：改进 Vector 类的 __eq__ 方法

```
def __eq__(self, other):
    if isinstance(other, Vector):  ❶
        return (len(self) == len(other) and
                all(a == b for a, b in zip(self, other)))
    else:
        return NotImplemented  ❷
```

❶ 如果 other 操作数是 Vector 实例（或者 Vector 子类的实例），那就像之前那样比较。

❷ 否则，返回 NotImplemented。

如果使用示例 13-13 中的新版 Vector.__eq__ 方法运行示例 13-12 中

的测试，得到的结果如示例 13-14 所示。

示例 **13-14**　与示例 13-12 一样的测试：最后一个结果变了

```
>>> va = Vector([1.0, 2.0, 3.0])
>>> vb = Vector(range(1, 4))
>>> va == vb  # ❶
True
>>> vc = Vector([1, 2])
>>> from vector2d_v3 import Vector2d
>>> v2d = Vector2d(1, 2)
>>> vc == v2d  # ❷
True
>>> t3 = (1, 2, 3)
>>> va == t3  # ❸
False
```

❶ 结果与之前一样，与预期相符。

❷ 结果与之前一样，但是为什么呢？稍后解释。[8]

> [8]这次不抛出异常，而是返回 `True`。请参阅前一个编者注。——编者注

❸ 结果不同了，这才是我们想要的。但是为什么会这样？请往下读……

在示例 13-14 中的三个结果里，第一个没变，但是后两个变了，这是因为示例 13-13 中的 `__eq__` 方法返回了 `NotImplemented`。`Vector` 实例与 `Vector2d` 实例比较时，具体步骤如下。

(1) 为了计算 `vc == v2d`，Python 调用 `Vector.__eq__(vc, v2d)`。

(2) 经 `Vector.__eq__(vc, v2d)` 确认，`v2d` 不是 `Vector` 实例，因此返回 `NotImplemented`。

(3) Python 得到 `NotImplemented` 结果，尝试调用 `Vector2d.__eq__(v2d, vc)`。

(4) `Vector2d.__eq__(v2d, vc)` 把两个操作数都变成元组，然后比较，结果是 `True`（`Vector2d.__eq__` 方法的代码在示例 9-9 中）。

在示例 13-14 中，`Vector` 实例和元组比较时，具体步骤如下。

(1) 为了计算 `va == t3`，Python 调用 `Vector.__eq__(va, t3)`。

(2) 经 `Vector.__eq__(va, t3)` 确认，`t3` 不是 `Vector` 实例，因此返回 `NotImplemented`。

(3) Python 得到 `NotImplemented` 结果，尝试调用 `tuple.__eq__(t3, va)`。

(4) `tuple.__eq__(t3, va)` 不知道 `Vector` 是什么，因此返回 `NotImplemented`。

(5) 对 `==` 来说，如果反向调用返回 `NotImplemented`，Python 会比较对象的 ID，作最后一搏。

那么 `!=` 运算符呢？我们不用实现它，因为从 `object` 继承的 `__ne__` 方法的后备行为满足了我们的需求：定义了 `__eq__` 方法，而且它不返回 `NotImplemented`，`__ne__` 会对 `__eq__` 返回的结果取反。

也就是说，对示例 13-14 中的对象来说，使用 `!=` 运算符比较的结果是一致的：

```
>>> va != vb
False
>>> vc != v2d
False
>>> va != (1, 2, 3)
True
```

从 `object` 中继承的 `__ne__` 方法，运作方式与下述代码类似，不过原版是用 C 语言实现的：[9]

> [9]`object.__eq__` 和 `object.__ne__` 的逻辑在 `object_richcompare` 函数中，位于 CPython 源码的 Objects/typeobject.c 文件（https://hg.python.org/cpython/file/c0e311e010fc/Objects/typeobject.c）中。

```
def __ne__(self, other):
    eq_result = self == other
```

```
    if eq_result is NotImplemented:
        return NotImplemented
    else:
        return not eq_result
```



Python 3 文档的缺陷 [10]

写作本书时，众多比较方法的文档（https://docs.python.org/3/reference/datamodel. html）说："`x==y` 成立不代表 `x!=y` 不成立。据此，如果定义 `__eq__()` 方法，也要定义 `__ne__()` 方法，这样运算符的行为才能符合预期。"对 Python 2 来说，确实是这样。但对 Python 3 而言，这不是好的建议，因为从 `object` 类继承的 `__ne__` 实现够用了，几乎不用重载。Guido 在他写的"What's New in Python 3.0"一文（https://docs.python.org/3/whatsnew/3.0.html#operators-and-special-methods）中说明了这个新行为，在"Operators And Special Methods"一节中。文档的这个缺陷在 issue 4395（http://bugs.python.org/issue4395）中做了记录。

[10]这个缺陷现在已经修正了。——编者注

讨论完重要的中缀运算符重载之后，下面换一类运算符：增量赋值运算符。

13.6　增量赋值运算符

`Vector` 类已经支持增量赋值运算符 `+=` 和 `*=` 了，如示例 13-15 所示。

示例 **13-15**　增量赋值不会修改不可变目标，而是新建实例，然后重新绑定

```
>>> v1 = Vector([1, 2, 3])
>>> v1_alias = v1  # ❶
>>> id(v1)  # ❷
4302860128
>>> v1 += Vector([4, 5, 6])  # ❸
>>> v1  # ❹
Vector([5.0, 7.0, 9.0])
>>> id(v1)  # ❺
4302859904
>>> v1_alias  # ❻
Vector([1.0, 2.0, 3.0])
>>> v1 *= 11  # ❼
>>> v1  # ❽
Vector([55.0, 77.0, 99.0])
>>> id(v1)
4302858336
```

❶ 复制一份，供后面审查 `Vector([1, 2, 3])` 对象。

❷ 记住一开始绑定给 `v1` 的 `Vector` 实例的 ID。

❸ 增量加法运算。

❹ 结果与预期相符……

❺ ……但是创建了新的 `Vector` 实例。

❻ 审查 `v1_alias`，确认原来的 `Vector` 实例没被修改。

❼ 增量乘法运算。

❽ 同样，结果与预期相符，但是创建了新的 `Vector` 实例。

如果一个类没有实现表 13-1 列出的就地运算符，增量赋值运算符只是语法糖：a += b 的作用与 a = a + b 完全一样。对不可变类型来说，这是预期的行为，而且，如果定义了 __add__ 方法的话，不用编写额外的代码，+= 就能使用。

然而，如果实现了就地运算符方法，例如 __iadd__，计算 a += b 的结果时会调用就地运算符方法。这种运算符的名称表明，它们会就地修改左操作数，而不会创建新对象作为结果。

> 不可变类型，如 Vector 类，一定不能实现就地特殊方法。这是明显的事实，不过还是值得提出来。

为了展示如何实现就地运算符，我们将扩展示例 11-12 中的 BingoCage 类，实现 __add__ 和 __iadd__ 方法。

我们把子类命名为 AddableBingoCage。示例 13-16 是我们想让 + 运算符具有的行为。

示例 **13-16**　使用 + 运算符新建 AddableBingoCage 实例

```
>>> vowels = 'AEIOU'
>>> globe = AddableBingoCage(vowels)  ❶
>>> globe.inspect()
('A', 'E', 'I', 'O', 'U')
>>> globe.pick() in vowels  ❷
True
>>> len(globe.inspect())  ❸
4
>>> globe2 = AddableBingoCage('XYZ')  ❹
>>> globe3 = globe + globe2
>>> len(globe3.inspect())  ❺
7
>>> void = globe + [10, 20]  ❻
Traceback (most recent call last):
  ...
TypeError: unsupported operand type(s) for +: 'AddableBingoCage' and 'list'
```

❶ 使用 5 个元素（vowels 中的各个字母）创建一个 globe 实例。

❷ 从中取出一个元素，确认它在 vowels 中。

❸ 确认 `globe` 的元素数量减少到 4 个了。

❹ 创建第二个实例，它有 3 个元素。

❺ 把前两个实例加在一起，创建第 3 个实例。这个实例有 7 个元素。

❻ `AddableBingoCage` 实例无法与列表相加，抛出 `TypeError`。那个错误消息是 `__add__` 方法返回 `NotImplemented` 时 Python 解释器输出的。

`AddableBingoCage` 是可变的，实现 `__iadd__` 方法后的行为如示例 13-17 所示。

示例 **13-17**　可以使用 `+=` 运算符载入现有的 `AddableBingoCage` 实例（接续示例 13-16）

```
>>> globe_orig = globe  ❶
>>> len(globe.inspect())  ❷
4
>>> globe += globe2  ❸
>>> len(globe.inspect())
7
>>> globe += ['M', 'N']  ❹
>>> len(globe.inspect())
9
>>> globe is globe_orig  ❺
True
>>> globe += 1  ❻
Traceback (most recent call last):
  ...
TypeError: right operand in += must be 'AddableBingoCage' or an iterable
```

❶ 复制一份，供后面检查对象的标识。

❷ 现在 `globe` 有 4 个元素。

❸ `AddableBingoCage` 实例可以从同属一类的其他实例那里接受元素。

❹ `+=` 的右操作数也可以是任何可迭代对象。

❺ 在这个示例中，globe 始终指代 globe_orig 对象。

❻ AddableBingoCage 实例不能与非可迭代对象相加，错误消息会指明原因。

注意，与 + 相比，+= 运算符对第二个操作数更宽容。+ 运算符的两个操作数必须是相同类型（这里是 AddableBingoCage），如若不然，结果的类型可能让人摸不着头脑。而 += 的情况更明确，因为就地修改左操作数，所以结果的类型是确定的。

> 通过观察内置 list 类型的工作方式，我确定了要对 + 和 += 的行为做什么限制。 my_list + x 只能用于把两个列表加到一起，而 my_list += x 可以使用右边可迭代对象 x 中的元素扩展左边的列表。list.extend() 的行为也是这样的，它的参数可以是任何可迭代对象。

我们明确了 AddableBingoCage 的行为，下面来看实现方式，如示例 13-18 所示。

示例 **13-18**　bingoaddable.py：AddableBingoCage 扩展 BingoCage，支持 + 和 +=

```
import itertools  ❶

from tombola import Tombola
from bingo import BingoCage


class AddableBingoCage(BingoCage):  ❷

    def __add__(self, other):
        if isinstance(other, Tombola):  ❸
            return AddableBingoCage(self.inspect() + other.inspect())  ❹
        else:
            return NotImplemented

    def __iadd__(self, other):
        if isinstance(other, Tombola):
            other_iterable = other.inspect()  ❺
        else:
```

```
        try:
            other_iterable = iter(other)
        except TypeError:  ❻
            self_cls = type(self).__name__
            msg = "right operand in += must be {!r} or an iterable"
            raise TypeError(msg.format(self_cls))
    self.load(other_iterable)  ❼
    return self  ❽
```

❶ “PEP 8—Style Guide for Python Code”（https://www.python.org/dev/peps/pep-0008/#imports）建议，把导入标准库的语句放在导入自己编写的模块之前。

❷ `AddableBingoCage` 扩展 `BingoCage`。

❸ `__add__` 方法的第二个操作数只能是 `Tombola` 实例。

❹ 如果 `other` 是 `Tombola` 实例，从中获取元素。

❺ 否则，尝试使用 `other` 创建迭代器。[11]

> [11]内置的 `iter` 函数在下一章讨论。这里，本可以使用 `tuple(other)`，这样做是可以的，但是 `.load(...)` 方法迭代参数时要构建大量元组，资源消耗大。

❻ 如果尝试失败，抛出异常，并且告知用户该怎么做。如果可能，错误消息应该明确指导用户怎么解决问题。

❼ 如果能执行到这里，把 `other_iterable` 载入 `self`。

❽ 重要提醒：增量赋值特殊方法必须返回 `self`。

通过示例 13-18 中 `__add__` 和 `__iadd__` 返回结果的方式可以总结出就地运算符的原理。

`__add__`

调用 `AddableBingoCage` 构造方法构建一个新实例，作为结果返回。

`__iadd__`

把修改后的 `self` 作为结果返回。

最后，示例 13-18 中还有一点要注意：从设计上看，`AddableBingoCage` 不用定义 `__radd__` 方法，因为不需要。如果右操作数是相同类型，那么正向方法 `__add__` 会处理，因此，Python 计算 `a + b` 时，如果 `a` 是 `AddableBingoCage` 实例，而 `b` 不是，那么会返回 `NotImplemented`，此时或许可以让 `b` 所属的类接手处理。可是，如果表达式是 `b + a`，而 `b` 不是 `AddableBingoCage` 实例，返回了 `NotImplemented`，那么 Python 最好放弃，抛出 `TypeError`，因为无法处理 `b`。

一般来说，如果中缀运算符的正向方法（如 `__mul__`）只处理与 `self` 属于同一类型的操作数，那就无需实现对应的反向方法（如 `__rmul__`），因为按照定义，反向方法是为了处理类型不同的操作数。

我们对 Python 运算符重载的讨论到此结束。

13.7　本章小结

本章首先说明了 Python 对运算符重载施加的一些限制：禁止重载内置类型的运算符，而且限于重载现有的运算符，不过有几个例外（`is`、`and`、`or`、`not`）。

随后，本章讲解了如何重载一元运算符，并实现了 `__neg__` 和 `__pos__` 方法。接着重载中缀运算符，首先是 `+`，它由 `__add__` 方法提供支持。我们得知，一元运算符和中缀运算符的结果应该是新对象，并且绝不能修改操作数。为了支持其他类型，我们返回特殊的 `NotImplemented` 值（不是异常），让解释器尝试对调操作数，然后调用运算符的反向特殊方法（如 `__radd__` ）。图 13-1 中的流程图概述了 Python 处理中缀运算符的算法。

如果操作数的类型不同，我们要检测出不能处理的操作数。本章使用两种方式处理这个问题：一种是鸭子类型，直接尝试执行运算，如果有问题，捕获 `TypeError` 异常；另一种是显式使用 `isinstance` 测试，`__mul__` 方法就是这么做的。这两种方式各有优缺点：鸭子类型更灵活，但是显式检查更能预知结果。如果选择使用 `isinstance`，要小心，不能测试具体类，而要测试 `numbers.Real` 抽象基类，例如 `isinstance(scalar, numbers.Real)`。这在灵活性和安全性之间做了很好的折中，因为当前或未来由用户定义的类型可以声明为抽象基类的真实子类或虚拟子类，详情参见第 11 章。

接下来的话题是众多比较运算符。我们通过 `__eq__` 方法实现了 `==`，而且发现 Python 在 `object` 基类中通过 `__ne__` 方法为 `!=` 提供了便利的实现。Python 处理这些运算符的方式与 `>`、`<`、`>=` 和 `<=` 稍有不同，具体而言是选择反向方法的逻辑不同，此外 Python 还会特别处理 `==` 和 `!=` 的后备机制：从不抛出错误，因为 Python 会比较对象的 ID，作最后一搏。

最后一节专门讨论了增量赋值运算符。我们发现，Python 处理这种运算符的方式是把它们当作常规的运算符加上赋值操作，即 `a += b` 其实会当成 `a = a + b` 处理。这样会始终创建新对象，因此可变类型和不可变类型都能用。对可变对象来说，可以实现就地特殊方法，例如支持

+= 的 __iadd__ 方法，然后修改左操作数的值。为了举例说明，我们把不可变的 Vector 类放到一边，为 BingoCage 的子类实现了 += 运算符，它会把元素添加到随机选号池中，这与内置的 list 类型把 += 当成 list.extend() 方法的快捷方式类似。在实现的过程中，我们得知在可接受的类型方面，+ 应该比 += 严格。对序列类型来说，+ 通常要求两个操作数属于同一类型，而 += 的右操作数往往可以是任何可迭代对象。

13.8　延伸阅读

在 Python 编程中，运算符重载经常使用 `isinstance` 做测试。一般来说，库应该利用动态类型（提高灵活性），避免显式测试类型，而是直接尝试操作，然后处理异常，这样只要对象支持所需的操作即可，而不必一定是某种类型。但是，Python 抽象基类允许一种更为严格的鸭子类型，Alex Martelli 称之为“白鹅类型”，编写重载运算符的代码时经常能用到。因此，如果你跳过了第 11 章，一定要去读读。

运算符特殊方法的主要参考资料是“Data Model”一章（https://docs.python.org/3/reference/datamodel.html）。这是权威资料，不过如“Python 3 文档的缺陷”所述，现在有个明显的缺陷，[12] 即建议“如果定义 `__eq__()` 方法，同时也要定义 `__ne__()` 方法”。实际上，在 Python 3 中，继承自 `object` 类的 `__ne__` 方法能满足绝大多数需求，因此一般很少实现 `__ne__` 方法。Python 标准库中 `numbers` 模块文档的“9.1.2.2. Implementing the arithmetic operations”一节（https://docs.python.org/3/library/numbers.html#implementing-the-arithmetic-operations）也值得一读。

> [12]这个缺陷现在已经修正了。——编者注

与之相关的一个技术是泛函数，由 Python 3 的 `@singledispatch` 装饰器支持（参见 7.8.2 节）。在 David Beazley 与 Brian K. Jones 的著作《Python Cookbook（第 3 版）中文版》中，“9.20 通过函数注解来实现方法重载”秘笈使用一些高级元编程（涉及元类）通过函数注解实现了基于类型的分派。Martelli、Ravenscroft 与 Ascher 的《Python Cookbook（第 2 版）中文版》一书有个有趣的诀窍（2.13 节，Erik Max Francis 提供），展示了如何重载 `<<` 运算符，在 Python 中模仿 C++ 的 `iostream` 句法。这两本书中还有一些其他关于运算符重载的示例，我只提了两个重要的诀窍。

`functools.total_ordering` 函数是个类装饰器（Python 2.7 及以上版本可用），它能为只定义了几个比较运算符的类自动生成全部比较运算符。请参阅 `functools` 模块的文档（https://docs.python.org/3/library/functools.html#functools.total_ordering）。

如果你对动态类型语言的运算符方法分派机制感兴趣，推荐阅读两篇具有重大意义的论文：Dan Ingalls（Smalltalk 团队的创始成员）写的“A Simple Technique for Handling Multiple Polymorphism”（https://wiki.illinois.edu//wiki/download/attachments/273416 以及 Kurt J. Hebel 与 Ralph Johnson（Johnson 是《设计模式：可复用面向对象软件的基础》的作者之一，因此出了名）合写的“Arithmetic and Double Dispatching in Smalltalk-80”（https://wiki.illinois.edu//wiki/download/attachments/273416327/double-dispatch.pdf）。这两篇论文深入分析了动态类型语言（如 Smalltalk、Python 和 Ruby）的多态。

Python 没有使用这两篇论文中所述的双重分配处理运算符。Python 使用的正向运算符和反向运算符更便于用户定义的类支持双重分派，但是这种方式需要解释器做些特殊处理。与之相比，经典的双重分派是一般性的技术，Python 和任何面向对象语言都能使用，而且不止适用于中缀运算符。其实，Ingalls、Hebel 和 Johnson 描述双重分派使用的示例完全不同。

本章开篇引用的那段话，以及“杂谈”中引用的两段话，均出自“The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling”一文（http://www.gotw.ca/publications/c_family_interview.htm），刊登于 *Java Report*, 5(7), July 2000 和 *C++ Report*, 12(7), July/August 2000 上。如果你对编程语言设计感兴趣，那么这篇文章非常值得一读。

杂谈

运算符重载的优缺点

如本章开头引用的那段话所述，James Gosling 决定不让 Java 支持运算符重载。在那次访谈中（“The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling”，http://www.gotw.ca/publications/c_family_interview.htm），他说：

> 大约 20% 到 30% 的人觉得运算符重载是罪恶之源；有些人对运算符的重载惹怒了很多人，因为他们使用 + 做列表插入，导致生活一团糟。这类问题大都源于一个事实：世界上有成千上

> 万个运算符，但是只有少数几个适合重载。因此，我们要挑选，但是有时所作的决定违背直觉。

Guido van Rossum 为运算符重载采取了一种折中方式：不放任用户随意创建运算符，如 `<=>` 或 `:-)`，这样防止了用户对运算符的异想天开，而且能让 Python 解析器保持简单。此外，Python 还禁止重载内置类型的运算符，这个限制也能增强可读性和可预知的性能。

Gosling 接着说道：

> 社区中约有 10% 的人能正确地使用和真正关心运算符重载，对这些人来说，运算符重载是极其重要的。这部分人几乎专门处理数字，在这一领域中，为了符合人类的直觉，表示法特别重要，因为他们进入这一领域时，直觉中已经知道 + 的意思，他们知道“a + b”中的 a 和 b 可以是复数、矩阵或其他合理的东西。

表示法方面的问题不能低估。下面以金融领域为例说明。在 Python 中，可以使用下述公式计算复利：

```
interest = principal * ((1 + rate) ** periods - 1)
```

不管涉及什么数字类型，这种表示法都成立。因此，如果是做重要的金融工作，你要确保 `periods` 是整数，`rate`、`interest` 和 `principal` 是精确的数字（Python 中 `decimal.Decimal` 类的实例），这样上述公式就能完好运行。

但是在 Java 中，如果把 `float` 换成精度不定的 `BigDecimal`，就无法再使用中缀运算符，因为中缀运算符只支持基本类型。在 Java 中，支持 `BigDecimal` 数字的公式要这样写：

```
BigDecimal interest = principal.multiply(BigDecimal.ONE.add(rate)
                          .pow(periods).subtract(BigDecimal.ONE));
```

显然，使用中缀运算符的公式更易读，至少对大多数人来说如此。[13] 为了让中缀运算符表示法支持非基本类型，运算符必须能重

载。Python 是门高级语言，易于使用，支持运算符重载可能就是它这些年在科学计算领域得到广泛使用的主要原因。

当然，语言不支持运算符重载也有好处。对极为重视性能和安全的低级系统语言而言，这无疑是正确的决定。新近出现的 Go 语言在这方面效仿了 Java，它不支持运算符重载。

但是，重载的运算符，如果使用得当，的确能让代码更易于阅读和编写。对现代的高级语言来说，这是个好功能。

惰性计算一瞥

如果仔细看示例 13-9 中的调用跟踪，会发现生成器表达式做惰性计算的证据。示例 13-19 再次列出那些调用跟踪，不过加上了一些标注。

示例 **13-19**　与示例 13-9 一样

```
>>> v1 + 'ABC'
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 File "vector_v6.py", line 329, in __add__
   return Vector(a + b for a, b in pairs)  # ❶
 File "vector_v6.py", line 243, in __init__
   self._components = array(self.typecode, components)  # ❷
 File "vector_v6.py", line 329, in <genexpr>
   return Vector(a + b for a, b in pairs)  # ❸
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

❶ `Vector` 调用的 `components` 参数是一个生成器表达式。这一步没问题。

❷ `components` 生成器表达式传给 `array` 构造方法。在这里，Python 尝试迭代生成器表达式，因此会计算第一个元素 `a + b`。这里抛出了 `TypeError`。

❸ 异常向上冒泡，到达 `Vector` 构造方法调用，在这里报告出来。

这表明，生成器表达式在最后时刻才会计算，而不是在源码中定义它的位置计算。

与之相比，如果像 `Vector([a + b for a, b in pairs])` 这样调用 `Vector` 构造方法，那么这里就会抛出异常，因为列表推导会尝试构建一个列表，以便作为参数传给 `Vector()` 调用。此时，根本不会触及 `Vector.__init__` 的定义体。

第 14 章会详细讨论生成器表达式，但是我不想让示例中偶然出现的惰性计算迹象漏过去。

[13]我的朋友 Mario Domenech Goulart，CHICKEN Scheme 编译器（http://www.call-cc.org）的核心开发者，可能不会同意这一说法。

第五部分　控制流程

第 14 章　可迭代的对象、迭代器和生成器

> 当我在自己的程序中发现用到了模式，我觉得这就表明某个地方出错了。程序的形式应该仅仅反映它所要解决的问题。代码中其他任何外加的形式都是一个信号，（至少对我来说）表明我对问题的抽象还不够深——这通常意味着自己正在手动完成的事情，本应该通过写代码来让宏的扩展自动实现。[1]
>
> ——Paul Graham[2]
> Lisp 黑客和风险投资人

> [1]摘自一篇博客文章，“Revenge of the Nerds”（“书呆子的复仇”，http://www.paulgraham.com/icad.html）。

> [2]Paul Graham 的文集《黑客与画家：来自计算机时代的高见》已由人民邮电出版社出版，书号：978-7-115-32656-0。——编者注

迭代是数据处理的基石。扫描内存中放不下的数据集时，我们要找到一种惰性获取数据项的方式，即按需一次获取一个数据项。这就是迭代器模式（Iterator pattern）。本章说明 Python 语言是如何内置迭代器模式的，这样就避免了自己手动去实现。

与 Lisp（Paul Graham 最喜欢的语言）不同，Python 没有宏，因此为了抽象出迭代器模式，需要改动语言本身。为此，Python 2.2（2001 年）加入了 `yield` 关键字。[3] 这个关键字用于构建生成器（generator），其作用与迭代器一样。

> [3]Python 2.2 的用户可以使用 `from __future__ import generators` 指令获取 `yield` 关键字；在 Python 2.3 中，`yield` 关键字默认可用。

所有生成器都是迭代器，因为生成器完全实现了迭代器接口。不过，根据《设计模式：可复用面向对象软件的基础》一书的

定义，迭代器用于从集合中取出元素；而生成器用于“凭空”生成元素。通过斐波纳契数列能很好地说明二者之间的区别：斐波纳契数列中的数有无穷个，在一个集合里放不下。不过要知道，在 Python 社区中，大多数时候都把迭代器和生成器视作同一概念。

在 Python 3 中，生成器有广泛的用途。现在，即使是内置的 `range()` 函数也返回一个类似生成器的对象，而以前则返回完整的列表。如果一定要让 `range()` 函数返回列表，那么必须明确指明（例如，`list(range(100))`）。

在 Python 中，所有集合都可以迭代。在 Python 语言内部，迭代器用于支持：

- `for` 循环
- 构建和扩展集合类型
- 逐行遍历文本文件
- 列表推导、字典推导和集合推导
- 元组拆包
- 调用函数时，使用 `*` 拆包实参

本章涵盖以下话题：

- 语言内部使用 `iter(...)` 内置函数处理可迭代对象的方式
- 如何使用 Python 实现经典的迭代器模式
- 详细说明生成器函数的工作原理
- 如何使用生成器函数或生成器表达式代替经典的迭代器
- 如何使用标准库中通用的生成器函数
- 如何使用 `yield from` 语句合并生成器

- 案例分析：在一个数据库转换工具中使用生成器函数处理大型数据集
- 为什么生成器和协程看似相同，实则差别很大，不能混淆

首先来研究 `iter(...)` 函数如何把序列变得可以迭代。

14.1　Sentence类第1版：单词序列

我们要实现一个 `Sentence` 类，以此打开探索可迭代对象的旅程。我们向这个类的构造方法传入包含一些文本的字符串，然后可以逐个单词迭代。第 1 版要实现序列协议，这个类的对象可以迭代，因为所有序列都可以迭代——这一点前面已经说过，不过现在要说明真正的原因。

示例 14-1 定义了一个 `Sentence` 类，通过索引从文本中提取单词。

示例 **14-1**　sentence.py：把句子划分为单词序列

```
import re
import reprlib

RE_WORD = re.compile('\w+')


class Sentence:

    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)   ❶

    def __getitem__(self, index):
        return self.words[index]   ❷

    def __len__(self):   ❸
        return len(self.words)

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)   ❹
```

❶ `re.findall` 函数返回一个字符串列表，里面的元素是正则表达式的全部非重叠匹配。

❷ `self.words` 中保存的是 `.findall` 函数返回的结果，因此直接返回指定索引位上的单词。

❸ 为了完善序列协议，我们实现了 `__len__` 方法；不过，为了让对象

可以迭代，没必要实现这个方法。

❹ reprlib.repr 这个实用函数用于生成大型数据结构的简略字符串表示形式。[4]

> [4]首次使用 reprlib 模块是在 10.2 节。

默认情况下，reprlib.repr 函数生成的字符串最多有 30 个字符。Sentence 类的用法参见示例 14-2 中的控制台会话。

示例 **14-2**　测试 Sentence 实例能否迭代

```
>>> s = Sentence('"The time has come," the Walrus said,')  # ❶
>>> s
Sentence('"The time ha... Walrus said,')  # ❷
>>> for word in s:  # ❸
...     print(word)
The
time
has
come
the
Walrus
said
>>> list(s)  # ❹
['The', 'time', 'has', 'come', 'the', 'Walrus', 'said']
```

❶ 传入一个字符串，创建一个 Sentence 实例。

❷ 注意，__repr__ 方法的输出中包含 reprlib.repr 方法生成的 ...。

❸ Sentence 实例可以迭代，稍后说明原因。

❹ 因为可以迭代，所以 Sentence 对象可以用于构建列表和其他可迭代的类型。

在接下来的几页中，我们还要开发其他 Sentence 类，而且都能通过示例 14-2 中的测试。不过，示例 14-1 中的实现与其他实现都不同，因为这一版 Sentence 类也是序列，可以按索引获取单词：

```
>>> s[0]
'The'
>>> s[5]
'Walrus'
>>> s[-1]
'said'
```

所有 Python 程序员都知道，序列可以迭代。下面说明具体的原因。

序列可以迭代的原因：**iter**函数

解释器需要迭代对象 x 时，会自动调用 `iter(x)`。

内置的 `iter` 函数有以下作用。

(1) 检查对象是否实现了 `__iter__` 方法，如果实现了就调用它，获取一个迭代器。

(2) 如果没有实现 `__iter__` 方法，但是实现了 `__getitem__` 方法，Python 会创建一个迭代器，尝试按顺序（从索引 0 开始）获取元素。

(3) 如果尝试失败，Python 抛出 `TypeError` 异常，通常会提示“C object is not iterable”（C 对象不可迭代），其中 C 是目标对象所属的类。

任何 Python 序列都可迭代的原因是，它们都实现了 `__getitem__` 方法。其实，标准的序列也都实现了 `__iter__` 方法，因此你也应该这么做。之所以对 `__getitem__` 方法做特殊处理，是为了向后兼容，而未来可能不会再这么做（不过，写作本书时还未弃用）。

11.2 节提到过，这是鸭子类型（duck typing）的极端形式：不仅要实现特殊的 `__iter__` 方法，还要实现 `__getitem__` 方法，而且 `__getitem__` 方法的参数是从 `0` 开始的整数（`int`），这样才认为对象是可迭代的。

在白鹅类型（goose-typing）理论中，可迭代对象的定义简单一些，不过没那么灵活：如果实现了 `__iter__` 方法，那么就认为对象是可迭代的。此时，不需要创建子类，也不用注册，因为 `abc.Iterable` 类实现了 `__subclasshook__` 方法，如 11.10 节所述。下面举个例子：

```
>>> class Foo:
...     def __iter__(self):
...         pass
...
>>> from collections import abc
>>> issubclass(Foo, abc.Iterable)
True
>>> f = Foo()
>>> isinstance(f, abc.Iterable)
True
```

不过要注意，虽然前面定义的 `Sentence` 类是可以迭代的，但却无法通过 `issubclass (Sentence, abc.Iterable)` 测试。

> 从 Python 3.4 开始，检查对象 `x` 能否迭代，最准确的方法是：调用 `iter(x)` 函数，如果不可迭代，再处理 `TypeError` 异常。这比使用 `isinstance(x, abc.Iterable)` 更准确，因为 `iter(x)` 函数会考虑到遗留的 `__getitem__` 方法，而 `abc.Iterable` 类则不考虑。

迭代对象之前显式检查对象是否可迭代或许没必要，毕竟尝试迭代不可迭代的对象时，Python 抛出的异常信息很明确：`TypeError: 'C' object is not iterable`。如果除了抛出 `TypeError` 异常之外还要做进一步的处理，可以使用 `try/except` 块，而无需显式检查。如果要保存对象，等以后再迭代，或许可以显式检查，因为这种情况可能需要尽早捕获错误。

下一节详述可迭代的对象和迭代器之间的关系。

14.2　可迭代的对象与迭代器的对比

从 14.1.1 节的解说可以推知下述定义。

可迭代的对象

　　使用 iter 内置函数可以获取迭代器的对象。如果对象实现了能返回迭代器的 __iter__ 方法，那么对象就是可迭代的。序列都可以迭代；实现了 __getitem__ 方法，而且其参数是从零开始的索引，这种对象也可以迭代。

我们要明确可迭代的对象和迭代器之间的关系：Python 从可迭代的对象中获取迭代器。

下面是一个简单的 for 循环，迭代一个字符串。这里，字符串 'ABC' 是可迭代的对象。背后是有迭代器的，只不过我们看不到：

```
>>> s = 'ABC'
>>> for char in s:
...     print(char)
...
A
B
C
```

如果没有 for 语句，不得不使用 while 循环模拟，要像下面这样写：

```
>>> s = 'ABC'
>>> it = iter(s)  # ❶
>>> while True:
...     try:
...         print(next(it))  # ❷
...     except StopIteration:  # ❸
...         del it  # ❹
...         break  # ❺
...
A
B
C
```

❶ 使用可迭代的对象构建迭代器 `it`。

❷ 不断在迭代器上调用 `next` 函数，获取下一个字符。

❸ 如果没有字符了，迭代器会抛出 `StopIteration` 异常。

❹ 释放对 `it` 的引用，即废弃迭代器对象。

❺ 退出循环。

`StopIteration` 异常表明迭代器到头了。Python 语言内部会处理 `for` 循环和其他迭代上下文（如列表推导、元组拆包，等等）中的 `StopIteration` 异常。

标准的迭代器接口有两个方法。

`__next__`

　　返回下一个可用的元素，如果没有元素了，抛出 `StopIteration` 异常。

`__iter__`

　　返回 `self`，以便在应该使用可迭代对象的地方使用迭代器，例如在 `for` 循环中。

这个接口在 `collections.abc.Iterator` 抽象基类中制定。这个类定义了 `__next__` 抽象方法，而且继承自 `Iterable` 类；`__iter__` 抽象方法则在 `Iterable` 类中定义。如图 14-1 所示。

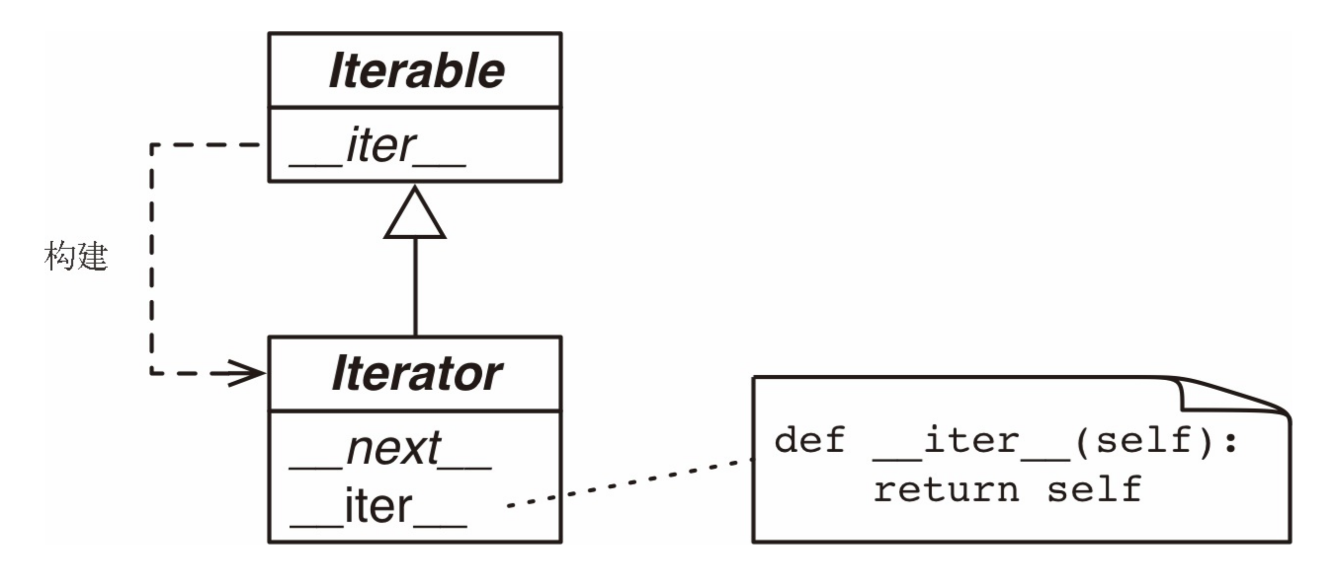


图 14-1：**Iterable** 和 **Iterator** 抽象基类。以斜体显示的是抽象方法。具体的 **Iterable.__iter__** 方法应该返回一个 **Iterator** 实例。具体的 **Iterator** 类必须实现 **__next__** 方法。**Iterator.__iter__** 方法直接返回实例本身

Iterator 抽象基类实现 __iter__ 方法的方式是返回实例本身（return self）。这样，在需要可迭代对象的地方可以使用迭代器。示例 14-3 是 abc.Iterator 类的源码。

示例 **14-3**　abc.Iterator 类，摘自 Lib/_collections_abc.py（https://hg.python.org/cpython/file/3.4/Lib/_collect

```
class Iterator(Iterable):

    __slots__ = ()

    @abstractmethod
    def __next__(self):
        'Return the next item from the iterator. When exhausted, raise StopIt
        raise StopIteration

    def __iter__(self):
        return self

    @classmethod
    def __subclasshook__(cls, C):
        if cls is Iterator:
            if (any("__next__" in B.__dict__ for B in C.__mro__) and
                any("__iter__" in B.__dict__ for B in C.__mro__)):
```

```
        return True
    return NotImplemented
```

在 Python 3 中，`Iterator` 抽象基类定义的抽象方法是 `it.__next__()`，而在 Python 2 中是 `it.next()`。一如既往，我们应该避免直接调用特殊方法，使用 `next(it)` 即可，这个内置的函数在 Python 2 和 Python 3 中都能使用。

在 Python 3.4 中，Lib/types.py（https://hg.python.org/cpython/file/3.4/Lib/types.py）模块的源码里有下面这段注释：

```
# Iterators in Python aren't a matter of type but of protocol.  A large
# and changing number of builtin types implement *some* flavor of
# iterator.  Don't check the type!  Use hasattr to check for both
# "__iter__" and "__next__" attributes instead.
```

其实，这就是 `abc.Iterator` 抽象基类中 `__subclasshook__` 方法的作用（参见示例 14-3）。

考虑到 Lib/types.py 中的建议，以及 Lib/_collections_abc.py 中的实现逻辑，检查对象 `x` 是否为迭代器最好的方式是调用 `isinstance(x, abc.Iterator)`。得益于 `Iterator.__subclasshook__` 方法，即使对象 `x` 所属的类不是 `Iterator` 类的真实子类或虚拟子类，也能这样检查。

再看示例 14-1 中定义的 `Sentence` 类，在 Python 控制台中能清楚地看出如何使用 `iter(...)` 函数构建迭代器，以及如何使用 `next(...)` 函数使用迭代器：

```
>>> s3 = Sentence('Pig and Pepper')  # ❶
>>> it = iter(s3)  # ❷
>>> it  # doctest: +ELLIPSIS
<iterator object at 0x...>
>>> next(it)  # ❸
'Pig'
```

```
>>> next(it)
'and'
>>> next(it)
'Pepper'
>>> next(it)  # ❹
Traceback (most recent call last):
  ...
StopIteration
>>> list(it)  # ❺
[]
>>> list(iter(s3))  # ❻
['Pig', 'and', 'Pepper']
```

❶ 创建一个 `Sentence` 实例 `s3`，包含 3 个单词。

❷ 从 `s3` 中获取迭代器。

❸ 调用 `next(it)`，获取下一个单词。

❹ 没有单词了，因此迭代器抛出 `StopIteration` 异常。

❺ 到头后，迭代器没用了。

❻ 如果想再次迭代，要重新构建迭代器。

因为迭代器只需 `__next__` 和 `__iter__` 两个方法，所以除了调用 `next()` 方法，以及捕获 `StopIteration` 异常之外，没有办法检查是否还有遗留的元素。此外，也没有办法“还原”迭代器。如果想再次迭代，那就要调用 `iter(...)`，传入之前构建迭代器的可迭代对象。传入迭代器本身没用，因为前面说过 `Iterator.__iter__` 方法的实现方式是返回实例本身，所以传入迭代器无法还原已经耗尽的迭代器。

根据本节的内容，可以得出迭代器的定义如下。

迭代器

　　迭代器是这样的对象：实现了无参数的 `__next__` 方法，返回序列中的下一个元素；如果没有元素了，那么抛出 `StopIteration` 异常。Python 中的迭代器还实现了 `__iter__` 方法，因此迭代器也可以迭代。

因为内置的 `iter(...)` 函数会对序列做特殊处理，所以第 1 版 `Sentence` 类可以迭代。接下来要实现标准的可迭代协议。

14.3 Sentence类第2版：典型的迭代器

第 2 版 Sentence 类根据《设计模式：可复用面向对象软件的基础》一书给出的模型，实现典型的迭代器设计模式。注意，这不符合 Python 的习惯做法，后面重构时会说明原因。不过，通过这一版能明确可迭代的集合和迭代器对象之间的关系。

示例 14-4 中定义的 Sentence 类可以迭代，因为它实现了特殊的 `__iter__` 方法，构建并返回一个 SentenceIterator 实例。《设计模式：可复用面向对象软件的基础》一书就是这样描述迭代器设计模式的。

这里之所以这么做，是为了清楚地说明可迭代的对象和迭代器之间的重要区别，以及二者之间的联系。

示例 **14-4**　sentence_iter.py：使用迭代器模式实现 Sentence 类

```
import re
import reprlib

RE_WORD = re.compile('\w+')


class Sentence:

    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)

    def __iter__(self):  ❶
        return SentenceIterator(self.words)  ❷


class SentenceIterator:

    def __init__(self, words):
        self.words = words  ❸
```

```
        self.index = 0  ❹

    def __next__(self):
        try:
            word = self.words[self.index]  ❺
        except IndexError:
            raise StopIteration()  ❻
        self.index += 1  ❼
        return word  ❽

    def __iter__(self):  ❾
        return self
```

❶ 与前一版相比，这里只多了一个 `__iter__` 方法。这一版没有 `__getitem__` 方法，为的是明确表明这个类可以迭代，因为实现了 `__iter__` 方法。

❷ 根据可迭代协议，`__iter__` 方法实例化并返回一个迭代器。

❸ `SentenceIterator` 实例引用单词列表。

❹ `self.index` 用于确定下一个要获取的单词。

❺ 获取 `self.index` 索引位上的单词。

❻ 如果 `self.index` 索引位上没有单词，那么抛出 `StopIteration` 异常。

❼ 递增 `self.index` 的值。

❽ 返回单词。

❾ 实现 `self.__iter__` 方法。

示例 14-4 中的代码能通过示例 14-2 中的测试。

注意，对这个示例来说，其实没必要在 `SentenceIterator` 类中实现 `__iter__` 方法，不过这么做是对的，因为迭代器应该实现 `__next__` 和 `__iter__` 两个方法，而且这么做能让迭代器通过

issubclass(SentenceInterator, abc.Iterator) 测试。如果让 SentenceIterator 类继承 abc.Iterator 类，那么它会继承 abc.Iterator.__iter__ 这个具体方法。

这一版的工作量很大（对懒惰的 Python 程序员来说确实如此）。注意，SentenceIterator 类的大多数代码都在处理迭代器的内部状态。稍后会说明如何简化。不过，在此之前我们先稍微离题，讨论一个看似合理实则错误的实现捷径。

把Sentence变成迭代器：坏主意

构建可迭代的对象和迭代器时经常会出现错误，原因是混淆了二者。要知道，可迭代的对象有个 __iter__ 方法，每次都实例化一个新的迭代器；而迭代器要实现 __next__ 方法，返回单个元素，此外还要实现 __iter__ 方法，返回迭代器本身。

因此，迭代器可以迭代，但是可迭代的对象不是迭代器。

除了 __iter__ 方法之外，你可能还想在 Sentence 类中实现 __next__ 方法，让 Sentence 实例既是可迭代的对象，也是自身的迭代器。可是，这种想法非常糟糕。根据有大量 Python 代码审查经验的 Alex Martelli 所说，这也是常见的反模式。

《设计模式：可复用面向对象软件的基础》一书讲解迭代器设计模式时，在“适用性”一节中说：[5]

> [5]《设计模式：可复用面向对象软件的基础》第 172 页。

> 迭代器模式可用来：
>
> - 访问一个聚合对象的内容而无需暴露它的内部表示
> - 支持对聚合对象的多种遍历
> - 为遍历不同的聚合结构提供一个统一的接口（即支持多态迭代）

为了“支持多种遍历”，必须能从同一个可迭代的实例中获取多个独立的迭代器，而且各个迭代器要能维护自身的内部状态，因此这一模式正确的实现方式是，每次调用 `iter(my_iterable)` 都新建一个独立的迭代器。这就是为什么这个示例需要定义 `SentenceIterator` 类。



可迭代的对象一定不能是自身的迭代器。也就是说，可迭代的对象必须实现 `__iter__` 方法，但不能实现 `__next__` 方法。

另一方面，迭代器应该一直可以迭代。迭代器的 `__iter__` 方法应该返回自身。

至此，我们演示了如何正确地实现典型的迭代器模式。本节至此告一段落，下一节展示如何使用更符合 Python 习惯的方式实现 `Sentence` 类。

14.4　Sentence类第3版：生成器函数

实现相同功能，但却符合 Python 习惯的方式是，用生成器函数代替 SentenceIterator 类。先看示例 14-5，然后详细说明生成器函数。

示例 **14-5**　sentence_gen.py：使用生成器函数实现 Sentence 类

```
import re
import reprlib

RE_WORD = re.compile('\w+')


class Sentence:

    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)

    def __iter__(self):
        for word in self.words:  ❶
            yield word  ❷
        return  ❸

# 完成！ ❹
```

❶ 迭代 self.words。

❷ 产出当前的 word。

❸ 这个 return 语句不是必要的；这个函数可以直接“落空”，自动返回。不管有没有 return 语句，生成器函数都不会抛出 StopIteration 异常，而是在生成完全部值之后会直接退出。[6]

> [6]Alex Martelli 审查这段代码时建议简化这个方法的定义体，直接使用 return iter(self.words)。当然，他是对的，毕竟调用 __iter__ 方法得到的就是迭代器。不过，

这里我用的是 for 循环，而且用到了 yield 关键字，这样做是为了介绍生成器函数的句法。下一节会详细说明。

❹ 不用再单独定义一个迭代器类！

我们又使用一种不同的方式实现了 Sentence 类，而且也能通过示例 14-2 中的测试。

在示例 14-4 定义的 Sentence 类中，__iter__ 方法调用 SentenceIterator 类的构造方法创建一个迭代器并将其返回。而在示例 14-5 中，迭代器其实是生成器对象，每次调用 __iter__ 方法都会自动创建，因为这里的 __iter__ 方法是生成器函数。

下面全面说明生成器函数。

生成器函数的工作原理

只要 Python 函数的定义体中有 yield 关键字，该函数就是生成器函数。调用生成器函数时，会返回一个生成器对象。也就是说，生成器函数是生成器工厂。

普通的函数与生成器函数在句法上唯一的区别是，在后者的定义体中有 yield 关键字。有些人认为定义生成器函数应该使用一个新的关键字，例如 gen，而不该使用 def，但是 Guido 不同意。他的理由参见“PEP 255—Simple Generators”（https://www.python.org/dev/peps/pep-0255/）。[7]

[7]有时，我会在生成器函数的名称中加上 gen 前缀或后缀，不过这不是习惯做法。显然，如果实现的是迭代器，那就不能这么做，因为所需的特殊方法必须命名为 __iter__。

下面以一个特别简单的函数说明生成器的行为：[8]

[8]感谢 David Kwast 建议使用这个示例。

```
>>> def gen_123():  # ❶
...     yield 1  # ❷
...     yield 2
```

```
...     yield 3
...
>>> gen_123  # doctest: +ELLIPSIS
<function gen_123 at 0x...>  # ❸
>>> gen_123()   # doctest: +ELLIPSIS
<generator object gen_123 at 0x...>  # ❹
>>> for i in gen_123():  # ❺
...     print(i)
1
2
3
>>> g = gen_123()  # ❻
>>> next(g)  # ❼
1
>>> next(g)
2
>>> next(g)
3
>>> next(g)  # ❽
Traceback (most recent call last):
  ...
StopIteration
```

❶ 只要 Python 函数中包含关键字 `yield`，该函数就是生成器函数。

❷ 生成器函数的定义体中通常都有循环，不过这不是必要条件；这里我重复使用 3 次 `yield`。

❸ 仔细看，`gen_123` 是函数对象。

❹ 但是调用时，`gen_123()` 返回一个生成器对象。

❺ 生成器是迭代器，会生成传给 `yield` 关键字的表达式的值。

❻ 为了仔细检查，我们把生成器对象赋值给 `g`。

❼ 因为 `g` 是迭代器，所以调用 `next(g)` 会获取 `yield` 生成的下一个元素。

❽ 生成器函数的定义体执行完毕后，生成器对象会抛出 `StopIteration` 异常。

生成器函数会创建一个生成器对象，包装生成器函数的定义体。把生成器传给 next(...) 函数时，生成器函数会向前，执行函数定义体中的下一个 yield 语句，返回产出的值，并在函数定义体的当前位置暂停。最终，函数的定义体返回时，外层的生成器对象会抛出 StopIteration 异常——这一点与迭代器协议一致。

我觉得，使用准确的词语描述从生成器中获取结果的过程，有助于理解生成器。注意，我说的是产出或生成值。如果说生成器“返回”值，就会让人难以理解。函数返回值；调用生成器函数返回生成器；生成器产出或生成值。生成器不会以常规的方式“返回”值：生成器函数定义体中的 return 语句会触发生成器对象抛出 StopIteration 异常。[9]

[9]在 Python 3.3 之前，如果生成器函数中的 return 语句有返回值，那么会报错。现在可以这么做，不过 return 语句仍会导致 StopIteration 异常抛出。调用方可以从异常对象中获取返回值。可是，只有把生成器函数当成协程使用时，这么做才有意义，详情参见 16.6 节。

示例 14-6 使用 for 循环更清楚地说明了生成器函数定义体的执行过程。

示例 **14-6**　运行时打印消息的生成器函数

```
>>> def gen_AB():  # ❶
...     print('start')
...     yield 'A'       # ❷
...     print('continue')
...     yield 'B'       # ❸
...     print('end.')   # ❹
...
>>> for c in gen_AB():  # ❺
...     print('-->', c)  # ❻
...
start   ❼
--> A   ❽
continue ❾
--> B   ❿
end.    ⓫
>>> ⓬
```

❶ 定义生成器函数的方式与普通的函数无异，只不过要使用 yield 关

键字。

❷ 在 for 循环中第一次隐式调用 next() 函数时（序号❺），会打印 'start'，然后停在第一个 yield 语句，生成值 'A'。

❸ 在 for 循环中第二次隐式调用 next() 函数时，会打印 'continue'，然后停在第二个 yield 语句，生成值 'B'。

❹ 第三次调用 next() 函数时，会打印 'end.'，然后到达函数定义体的末尾，导致生成器对象抛出 StopIteration 异常。

❺ 迭代时，for 机制的作用与 g = iter(gen_AB()) 一样，用于获取生成器对象，然后每次迭代时调用 next(g)。

❻ 循环块打印 --> 和 next(g) 返回的值。但是，生成器函数中的 print 函数输出结果之后才会看到这个输出。

❼ 'start' 是生成器函数定义体中 print('start') 输出的结果。

❽ 生成器函数定义体中的 yield 'A' 语句会生成值 A，提供给 for 循环使用，而 A 会赋值给变量 c，最终输出 --> A。

❾ 第二次调用 next(g)，继续迭代，生成器函数定义体中的代码由 yield 'A' 前进到 yield 'B'。文本 continue 是由生成器函数定义体中的第二个 print 函数输出的。

❿ yield 'B' 语句生成值 B，提供给 for 循环使用，而 B 会赋值给变量 c，所以循环打印出 --> B。

⓫ 第三次调用 next(it)，继续迭代，前进到生成器函数的末尾。文本 end. 是由生成器函数定义体中的第三个 print 函数输出的。到达生成器函数定义体的末尾时，生成器对象抛出 StopIteration 异常。for 机制会捕获异常，因此循环终止时没有报错。

⓬ 现在，希望你已经知道示例 14-5 中 Sentence.__iter__ 方法的作用了：__iter__ 方法是生成器函数，调用时会构建一个实现了迭代器接口的生成器对象，因此不用再定义 SentenceIterator 类了。

这一版 `Sentence` 类比前一版简短多了，但是还不够懒惰。如今，人们认为惰性是好的特质，至少在编程语言和 API 中是如此。惰性实现是指尽可能延后生成值。这样做能节省内存，而且或许还可以避免做无用的处理。

下一节以这种惰性方式定义 `Sentence` 类。

14.5　Sentence类第4版：惰性实现

设计 Iterator 接口时考虑到了惰性：next(my_iterator) 一次生成一个元素。懒惰的反义词是急迫，其实，惰性求值（lazy evaluation）和及早求值（eager evaluation）是编程语言理论方面的技术术语。

目前实现的几版 Sentence 类都不具有惰性，因为 __init__ 方法急迫地构建好了文本中的单词列表，然后将其绑定到 self.words 属性上。这样就得处理整个文本，列表使用的内存量可能与文本本身一样多（或许更多，这取决于文本中有多少非单词字符）。如果只需迭代前几个单词，大多数工作都是白费力气。

只要使用的是 Python 3，思索着做某件事有没有懒惰的方式，答案通常都是肯定的。

re.finditer 函数是 re.findall 函数的惰性版本，返回的不是列表，而是一个生成器，按需生成 re.MatchObject 实例。如果有很多匹配，re.finditer 函数能节省大量内存。我们要使用这个函数让第 4 版 Sentence 类变得懒惰，即只在需要时才生成下一个单词。代码如示例 14-7 所示。

示例 **14-7**　sentence_gen2.py：在生成器函数中调用 re.finditer 生成器函数，实现 Sentence 类

```
import re
import reprlib

RE_WORD = re.compile('\w+')


class Sentence:

    def __init__(self, text):
        self.text = text  ❶

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)
```

```
    def __iter__(self):
        for match in RE_WORD.finditer(self.text):  ❷
            yield match.group()  ❸
```

❶ 不再需要 words 列表。

❷ finditer 函数构建一个迭代器，包含 self.text 中匹配 RE_WORD 的单词，产出 MatchObject 实例。

❸ match.group() 方法从 MatchObject 实例中提取匹配正则表达式的具体文本。

生成器函数已经极大地简化了代码，但是使用生成器表达式甚至能把代码变得更简短。

14.6　Sentence类第5版：生成器表达式

简单的生成器函数，如前面的 Sentence 类中使用的那个（见示例 14-7），可以替换成生成器表达式。

生成器表达式可以理解为列表推导的惰性版本：不会迫切地构建列表，而是返回一个生成器，按需惰性生成元素。也就是说，如果列表推导是制造列表的工厂，那么生成器表达式就是制造生成器的工厂。

示例 14-8 演示了一个简单的生成器表达式，并且与列表推导做了对比。

示例 **14-8**　先在列表推导中使用 gen_AB 生成器函数，然后在生成器表达式中使用

```
>>> def gen_AB():  # ❶
...     print('start')
...     yield 'A'
...     print('continue')
...     yield 'B'
...     print('end.')
...
>>> res1 = [x*3 for x in gen_AB()]  # ❷
start
continue
end.
>>> for i in res1:  # ❸
...     print('-->', i)
...
--> AAA
--> BBB
>>> res2 = (x*3 for x in gen_AB())  # ❹
>>> res2  # ❺
<generator object <genexpr> at 0x10063c240>
>>> for i in res2:  # ❻
...     print('-->', i)
...
start
--> AAA
continue
--> BBB
```

```
end.
```

❶ gen_AB 函数与示例 14-6 中的一样。

❷ 列表推导迫切地迭代 gen_AB() 函数生成的生成器对象产出的元素：'A' 和 'B'。注意，下面的输出是 start、continue 和 end.。

❸ 这个 for 循环迭代列表推导生成的 res1 列表。

❹ 把生成器表达式返回的值赋值给 res2。只需调用 gen_AB() 函数，虽然调用时会返回一个生成器，但是这里并不使用。

❺ res2 是一个生成器对象。

❻ 只有 for 循环迭代 res2 时，gen_AB 函数的定义体才会真正执行。for 循环每次迭代时会隐式调用 next(res2)，前进到 gen_AB 函数中的下一个 yield 语句。注意，gen_AB 函数的输出与 for 循环中 print 函数的输出夹杂在一起。

可以看出，生成器表达式会产出生成器，因此可以使用生成器表达式进一步减少 Sentence 类的代码，如示例 14-9 所示。

示例 **14-9**　sentence_genexp.py：使用生成器表达式实现 Sentence 类

```
import re
import reprlib

RE_WORD = re.compile('\w+')


class Sentence:

    def __init__(self, text):
      self.text = text

    def __repr__(self):
      return 'Sentence(%s)' % reprlib.repr(self.text)

    def __iter__(self):
      return (match.group() for match in RE_WORD.finditer(self.text))
```

与示例 14-7 唯一的区别是 `__iter__` 方法，这里不是生成器函数了（没有 `yield`），而是使用生成器表达式构建生成器，然后将其返回。不过，最终的效果一样：调用 `__iter__` 方法会得到一个生成器对象。

生成器表达式是语法糖：完全可以替换成生成器函数，不过有时使用生成器表达式更便利。下一节说明生成器表达式的用途。

14.7　何时使用生成器表达式

在示例 10-16 中，为了实现 Vector 类，我用了几个生成器表达式，__eq__、__hash__、__abs__、angle、angles、format、__add__ 和 __mul__ 方法中各有一个生成器表达式。在这些方法中使用列表推导也行，不过立即返回的列表要使用更多的内存。

通过示例 14-9 可知，生成器表达式是创建生成器的简洁句法，这样无需先定义函数再调用。不过，生成器函数灵活得多，可以使用多个语句实现复杂的逻辑，也可以作为协程使用（参见第 16 章）。

遇到简单的情况时，可以使用生成器表达式，因为这样扫一眼就知道代码的作用，如 Vector 类的示例所示。

根据我的经验，选择使用哪种句法很容易判断：如果生成器表达式要分成多行写，我倾向于定义生成器函数，以便提高可读性。此外，生成器函数有名称，因此可以重用。



句法提示

如果函数或构造方法只有一个参数，传入生成器表达式时不用写一对调用函数的括号，再写一对括号围住生成器表达式，只写一对括号就行了，如示例 10-16 中 __mul__ 方法对 Vector 构造方法的调用，转摘如下。然而，如果生成器表达式后面还有其他参数，那么必须使用括号围住，否则会抛出 SyntaxError 异常：

```
def __mul__(self, scalar):
    if isinstance(scalar, numbers.Real):
        return Vector(n * scalar for n in self)
    else:
        return NotImplemented
```

目前所见的 Sentence 类示例说明了如何把生成器当作典型的迭代器使用，即从集合中获取元素。不过，生成器也可用于生成不受数据源限制

的值。下一节会举例说明。

14.8　另一个示例：等差数列生成器

典型的迭代器模式作用很简单——遍历数据结构。不过，即便不是从集合中获取元素，而是获取序列中即时生成的下一个值时，也用得到这种基于方法的标准接口。例如，内置的 `range` 函数用于生成有穷整数等差数列（Arithmetic Progression，AP），`itertools.count` 函数用于生成无穷等差数列。

下一节会说明 `itertools.count` 函数，本节探讨如何生成不同数字类型的有穷等差数列。

下面我们在控制台中对稍后实现的 `ArithmeticProgression` 类做一些测试，如示例 14-10 所示。这里，构造方法的签名是 `ArithmeticProgression(begin, step[, end])`。`range()` 函数与这个 `ArithmeticProgression` 类的作用类似，不过签名是 `range(start, stop[, step])`。我选择使用不同的签名是因为，创建等差数列时必须指定公差（`step`），而末项（`end`）是可选的。我还把参数的名称由 `start/stop` 改成了 `begin/end`，以明确表明签名不同。在示例 14-10 里的每个测试中，我都调用了 `list()` 函数，用于查看生成的值。

示例 **14-10**　演示 `ArithmeticProgression` 类的用法

```
>>> ap = ArithmeticProgression(0, 1, 3)
>>> list(ap)
[0, 1, 2]
>>> ap = ArithmeticProgression(1, .5, 3)
>>> list(ap)
[1.0, 1.5, 2.0, 2.5]
>>> ap = ArithmeticProgression(0, 1/3, 1)
>>> list(ap)
[0.0, 0.3333333333333333, 0.6666666666666666]
>>> from fractions import Fraction
>>> ap = ArithmeticProgression(0, Fraction(1, 3), 1)
>>> list(ap)
[Fraction(0, 1), Fraction(1, 3), Fraction(2, 3)]
>>> from decimal import Decimal
>>> ap = ArithmeticProgression(0, Decimal('.1'), .3)
```

```
>>> list(ap)
[Decimal('0.0'), Decimal('0.1'), Decimal('0.2')]
```

注意，在得到的等差数列中，数字的类型与 begin 或 step 的类型一致。如果需要，会根据 Python 算术运算的规则强制转换类型。在示例 14-10 中，有 int、float、Fraction 和 Decimal 数字组成的列表。

示例 14-11 列出的是 ArithmeticProgression 类的实现。

示例 **14-11**　ArithmeticProgression 类

```
class ArithmeticProgression:

    def __init__(self, begin, step, end=None):  ❶
        self.begin = begin
        self.step = step
        self.end = end  # None -> 无穷数列

    def __iter__(self):
        result = type(self.begin + self.step)(self.begin)  ❷
        forever = self.end is None  ❸
        index = 0
        while forever or result < self.end:  ❹
            yield result  ❺
            index += 1
            result = self.begin + self.step * index  ❻
```

❶ __init__ 方法需要两个参数：begin 和 step。end 是可选的，如果值是 None，那么生成的是无穷数列。

❷ 这一行把 self.begin 赋值给 result，不过会先强制转换成前面的加法算式得到的类型。[10]

[10]Python 2 内置了 coerce() 函数，不过 Python 3 没有内置。开发者觉得没必要内置，因为算术运算符会隐式应用数值强制转换规则。所以，为了让数列的首项与其他项的类型一样，我能想到最好的方式是，先做加法运算，然后使用计算结果的类型强制转换生成的结果。我在 Python 邮件列表中问了这个问题，Steven D'Aprano 给出了妙极的答复（https://mail.python.org/pipermail/python-list/2014-December/682651.html）。

❸ 为了提高可读性，我们创建了 forever 变量，如果 self.end 属性

的值是 None，那么 forever 的值是 True，因此生成的是无穷数列。

❹ 这个循环要么一直执行下去，要么当 result 大于或等于 self.end 时结束。如果循环退出了，那么这个函数也随之退出。

❺ 生成当前的 result 值。

❻ 计算可能存在的下一个结果。这个值可能永远不会产出，因为 while 循环可能会终止。

在示例 14-11 中的最后一行，我没有直接使用 self.step 不断地增加 result，而是选择使用 index 变量，把 self.begin 与 self.step 和 index 的乘积相加，计算 result 的各个值，以此降低处理浮点数时累积效应致错的风险。

示例 14-11 中定义的 ArithmeticProgression 类能按预期那样使用。这是个简单的示例，说明了如何使用生成器函数实现特殊的 __iter__ 方法。然而，如果一个类只是为了构建生成器而去实现 __iter__ 方法，那还不如使用生成器函数。毕竟，生成器函数是制造生成器的工厂。

示例 14-12 中定义了一个名为 aritprog_gen 的生成器函数，作用与 ArithmeticProgression 类一样，只不过代码量更少。如果把 ArithmeticProgression 类换成 aritprog_gen 函数，示例 14-10 中的测试也都能通过。[11]

[11]本书源码仓库（https://github.com/fluentpython/example-code）中的 14-it-generator/ 目录里包含 doctest，以及一个 aritprog_runner.py 脚本，用于测试 aritprog*.py 脚本的所有版本。

示例 **14-12**　aritprog_gen 生成器函数

```
def aritprog_gen(begin, step, end=None):
    result = type(begin + step)(begin)
    forever = end is None
    index = 0
    while forever or result < end:
        yield result
        index += 1
        result = begin + step * index
```

示例 14-12 很棒，不过始终要记住，标准库中有许多现成的生成器。下一节会使用 `itertools` 模块实现，那个版本更棒。

使用itertools模块生成等差数列

Python 3.4 中的 `itertools` 模块提供了 19 个生成器函数，结合起来使用能实现很多有趣的用法。

例如，`itertools.count` 函数返回的生成器能生成多个数。如果不传入参数，`itertools.count` 函数会生成从零开始的整数数列。不过，我们可以提供可选的 `start` 和 `step` 值，这样实现的作用与 `aritprog_gen` 函数十分相似：

```
>>> import itertools
>>> gen = itertools.count(1, .5)
>>> next(gen)
1
>>> next(gen)
1.5
>>> next(gen)
2.0
>>> next(gen)
2.5
```

然而，`itertools.count` 函数从不停止，因此，如果调用 `list(count())`，Python 会创建一个特别大的列表，超出可用内存，在调用失败之前，电脑会疯狂地运转。

不过，`itertools.takewhile` 函数则不同，它会生成一个使用另一个生成器的生成器，在指定的条件计算结果为 `False` 时停止。因此，可以把这两个函数结合在一起使用，编写下述代码：

```
>>> gen = itertools.takewhile(lambda n: n < 3, itertools.count(1, .5))
>>> list(gen)
[1, 1.5, 2.0, 2.5]
```

示例 14-13 利用 `takewhile` 和 `count` 函数，写出的代码流畅而简短。

示例 **14-13**　aritprog_v3.py：与前面的 `aritprog_gen` 函数作用相同

```
import itertools


def aritprog_gen(begin, step, end=None):
    first = type(begin + step)(begin)
    ap_gen = itertools.count(first, step)
    if end is not None:
        ap_gen = itertools.takewhile(lambda n: n < end, ap_gen)
    return ap_gen
```

注意，示例 14-13 中的 `aritprog_gen` 不是生成器函数，因为定义体中没有 `yield` 关键字。但是它会返回一个生成器，因此它与其他生成器函数一样，也是生成器工厂函数。

示例 14-13 想表达的观点是，实现生成器时要知道标准库中有什么可用，否则很可能会重新发明轮子。鉴于此，下一节会介绍一些现成的生成器函数。

14.9　标准库中的生成器函数

标准库提供了很多生成器，有用于逐行迭代纯文本文件的对象，还有出色的 `os.walk` 函数（https://docs.python.org/3/library/os.html#os.walk）。这个函数在遍历目录树的过程中产出文件名，因此递归搜索文件系统像 `for` 循环那样简单。

`os.walk` 生成器函数的作用令人赞叹，不过本节专注于通用的函数：参数为任意的可迭代对象，返回值是生成器，用于生成选中的、计算出的和重新排列的元素。在下述几个表格中，我会概述其中的 24 个，有些是内置的，有些在 `itertools` 和 `functools` 模块中。为了方便，我按照函数的高阶功能分组，而不管函数是在哪里定义的。

> 你可能知道本节所述的全部函数，但是某些函数没有得到充分利用，因此快速概览一遍能让你知道有什么函数可用。

第一组是用于过滤的生成器函数：从输入的可迭代对象中产出元素的子集，而且不修改元素本身。本章前面的 14.8.1 节用过 `itertools.takewhile` 函数。与 `takewhile` 函数一样，表 14-1 中的大多数函数都接受一个断言参数（`predicate`）。这个参数是个布尔函数，有一个参数，会应用到输入中的每个元素上，用于判断元素是否包含在输出中。

表**14-1**：用于过滤的生成器函数

模块	函数	说明
`itertools`	`compress(it, selector_it)`	并行处理两个可迭代的对象；如果 `selector_it` 中的元素是真值，产出 `it` 中对应的元素
`itertools`	`dropwhile(predicate, it)`	处理 `it`，跳过 `predicate` 的计算结果为真值的元素，然后产出剩下的各个元素（不再进一步检查）

（内置）	filter(predicate, it)	把 it 中的各个元素传给 predicate，如果 predicate(item) 返回真值，那么产出对应的元素；如果 predicate 是 None，那么只产出真值元素
itertools	filterfalse(predicate, it)	与 filter 函数的作用类似，不过 predicate 的逻辑是相反的：predicate 返回假值时产出对应的元素
itertools	islice(it, stop) 或 islice(it, start, stop, step=1)	产出 it 的切片，作用类似于 s[:stop] 或 s[start:stop:step]，不过 it 可以是任何可迭代的对象，而且这个函数实现的是惰性操作
itertools	takewhile(predicate, it)	predicate 返回真值时产出对应的元素，然后立即停止，不再继续检查

示例 14-14 在控制台中演示表 14-1 中各个函数的用法。

示例 **14-14**　演示用于过滤的生成器函数

```
>>> def vowel(c):
...     return c.lower() in 'aeiou'
...
>>> list(filter(vowel, 'Aardvark'))
['A', 'a', 'a']
>>> import itertools
>>> list(itertools.filterfalse(vowel, 'Aardvark'))
['r', 'd', 'v', 'r', 'k']
>>> list(itertools.dropwhile(vowel, 'Aardvark'))
['r', 'd', 'v', 'a', 'r', 'k']
>>> list(itertools.takewhile(vowel, 'Aardvark'))
['A', 'a']
>>> list(itertools.compress('Aardvark', (1,0,1,1,0,1)))
['A', 'r', 'd', 'a']
>>> list(itertools.islice('Aardvark', 4))
['A', 'a', 'r', 'd']
>>> list(itertools.islice('Aardvark', 4, 7))
['v', 'a', 'r']
>>> list(itertools.islice('Aardvark', 1, 7, 2))
['a', 'd', 'a']
```

下一组是用于映射的生成器函数：在输入的单个可迭代对象（`map` 和 `starmap` 函数处理多个可迭代的对象）中的各个元素上做计算，然后返回结果。[12] 表 14-2 中的生成器函数会从输入的可迭代对象中的各个元素中产出一个元素。如果输入来自多个可迭代的对象，第一个可迭代的对象到头后就停止输出。

> [12]这里所说的“映射”与字典没有关系，而与内置的 `map` 函数有关。

表**14-2**：用于映射的生成器函数

模块	函数	说明
itertools	accumulate(it, [func])	产出累积的总和；如果提供了 `func`，那么把前两个元素传给它，然后把计算结果和下一个元素传给它，以此类推，最后产出结果
（内置）	enumerate(iterable, start=0)	产出由两个元素组成的元组，结构是 `(index, item)`，其中 `index` 从 `start` 开始计数，`item` 则从 `iterable` 中获取
（内置）	map(func, it1, [it2, ..., itN])	把 `it` 中的各个元素传给func，产出结果；如果传入 *N* 个可迭代的对象，那么 `func` 必须能接受 *N* 个参数，而且要并行处理各个可迭代的对象
itertools	starmap(func, it)	把 `it` 中的各个元素传给 `func`，产出结果；输入的可迭代对象应该产出可迭代的元素 `iit`，然后以 `func(*iit)` 这种形式调用 `func`

示例 14-15 演示 `itertools.accumulate` 函数的几个用法。

示例 **14-15**　演示 `itertools.accumulate` 生成器函数

```
>>> sample = [5, 4, 2, 8, 7, 6, 3, 0, 9, 1]
>>> import itertools
>>> list(itertools.accumulate(sample))  # ❶
[5, 9, 11, 19, 26, 32, 35, 35, 44, 45]
```

```
>>> list(itertools.accumulate(sample, min))  # ❷
[5, 4, 2, 2, 2, 2, 2, 0, 0, 0]
>>> list(itertools.accumulate(sample, max))  # ❸
[5, 5, 5, 8, 8, 8, 8, 8, 9, 9]
>>> import operator
>>> list(itertools.accumulate(sample, operator.mul))  # ❹
[5, 20, 40, 320, 2240, 13440, 40320, 0, 0, 0]
>>> list(itertools.accumulate(range(1, 11), operator.mul))
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]  # ❺
```

❶ 计算总和。

❷ 计算最小值。

❸ 计算最大值。

❹ 计算乘积。

❺ 从 `1!` 到 `10!`，计算各个数的阶乘。

表 14-2 中剩余函数的演示如示例 14-16 所示。

示例 **14-16**　演示用于映射的生成器函数

```
>>> list(enumerate('albatroz', 1))  # ❶
[(1, 'a'), (2, 'l'), (3, 'b'), (4, 'a'), (5, 't'), (6, 'r'), (7, 'o'), (8, 'z
>>> import operator
>>> list(map(operator.mul, range(11), range(11)))  # ❷
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> list(map(operator.mul, range(11), [2, 4, 8]))  # ❸
[0, 4, 16]
>>> list(map(lambda a, b: (a, b), range(11), [2, 4, 8]))  # ❹
[(0, 2), (1, 4), (2, 8)]
>>> import itertools
>>> list(itertools.starmap(operator.mul, enumerate('albatroz', 1)))  # ❺
['a', 'll', 'bbb', 'aaaa', 'ttttt', 'rrrrrr', 'ooooooo', 'zzzzzzzz']
>>> sample = [5, 4, 2, 8, 7, 6, 3, 0, 9, 1]
>>> list(itertools.starmap(lambda a, b: b/a,
...     enumerate(itertools.accumulate(sample), 1)))  # ❻
[5.0, 4.5, 3.6666666666666665, 4.75, 5.2, 5.333333333333333,
5.0, 4.375, 4.888888888888889, 4.5]
```

❶ 从 `1` 开始，为单词中的字母编号。

❷ 从 `0` 到 `10`，计算各个整数的平方。

❸ 计算两个可迭代对象中对应位置上的两个元素之积，元素最少的那个可迭代对象到头后就停止。

❹ 作用等同于内置的 `zip` 函数。

❺ 从 `1` 开始，根据字母所在的位置，把字母重复相应的次数。

❻ 计算平均值。

接下来这一组是用于合并的生成器函数，这些函数都从输入的多个可迭代对象中产出元素。`chain` 和 `chain.from_iterable` 按顺序（一个接一个）处理输入的可迭代对象，而 `product`、`zip` 和 `zip_longest` 并行处理输入的各个可迭代对象。如表 14-3 所示。

表**14-3**：合并多个可迭代对象的生成器函数

模块	函数	说明
`itertools`	`chain(it1, ..., itN)`	先产出 `it1` 中的所有元素，然后产出 `it2` 中的所有元素，以此类推，无缝连接在一起
`itertools`	`chain.from_iterable(it)`	产出 `it` 生成的各个可迭代对象中的元素，一个接一个，无缝连接在一起；`it` 应该产出可迭代的元素，例如可迭代的对象列表
`itertools`	`product(it1, ..., itN, repeat=1)`	计算笛卡儿积：从输入的各个可迭代对象中获取元素，合并成由 *N* 个元素组成的元组，与嵌套的 `for` 循环效果一样；`repeat` 指明重复处理多少次输入的可迭代对象
（内置）	`zip(it1, ..., itN)`	并行从输入的各个可迭代对象中获取元素，产出由 *N* 个元素组成的元组，只要有一个可迭代的对象到头了，就默默地停止

itertools	zip_longest(it1, ..., itN, fillvalue=None)	并行从输入的各个可迭代对象中获取元素，产出由 *N* 个元素组成的元组，等到最长的可迭代对象到头后才停止，空缺的值使用 fillvalue 填充

示例 14-17 展示 `itertools.chain` 和 `zip` 生成器函数及其同胞的用法。再次提醒，`zip` 函数的名称出自 zip fastener 或 zipper（拉链，与 ZIP 压缩没有关系）。“出色的 `zip` 函数”附注栏介绍过 `zip` 和 `itertools.zip_longest` 函数。

示例 **14-17** 演示用于合并的生成器函数

```
>>> list(itertools.chain('ABC', range(2)))  # ❶
['A', 'B', 'C', 0, 1]
>>> list(itertools.chain(enumerate('ABC')))  # ❷
[(0, 'A'), (1, 'B'), (2, 'C')]
>>> list(itertools.chain.from_iterable(enumerate('ABC')))  # ❸
[0, 'A', 1, 'B', 2, 'C']
>>> list(zip('ABC', range(5)))  # ❹
[('A', 0), ('B', 1), ('C', 2)]
>>> list(zip('ABC', range(5), [10, 20, 30, 40]))  # ❺
[('A', 0, 10), ('B', 1, 20), ('C', 2, 30)]
>>> list(itertools.zip_longest('ABC', range(5)))  # ❻
[('A', 0), ('B', 1), ('C', 2), (None, 3), (None, 4)]
>>> list(itertools.zip_longest('ABC', range(5), fillvalue='?'))  # ❼
[('A', 0), ('B', 1), ('C', 2), ('?', 3), ('?', 4)]
```

❶ 调用 `chain` 函数时通常传入两个或更多个可迭代对象。

❷ 如果只传入一个可迭代的对象，那么 `chain` 函数没什么用。

❸ 但是 `chain.from_iterable` 函数从可迭代的对象中获取每个元素，然后按顺序把元素连接起来，前提是各个元素本身也是可迭代的对象。

❹ `zip` 常用于把两个可迭代的对象合并成一系列由两个元素组成的元组。

❺ `zip` 可以并行处理任意数量个可迭代的对象，不过只要有一个可迭代的对象到头了，生成器就停止。

❻ itertools.zip_longest 函数的作用与 zip 类似，不过输入的所有可迭代对象都会处理到头，如果需要会填充 None。

❼ fillvalue 关键字参数用于指定填充的值。

itertools.product 生成器是计算笛卡儿积的惰性方式；在 2.2.3 节，我们在多个 for 子句中使用列表推导计算过笛卡儿积。此外，也可以使用包含多个 for 子句的生成器表达式，以惰性方式计算笛卡儿积。示例 14-18 演示 itertools.product 函数的用法。

示例 **14-18**　演示 itertools.product 生成器函数

```
>>> list(itertools.product('ABC', range(2)))  # ❶
[('A', 0), ('A', 1), ('B', 0), ('B', 1), ('C', 0), ('C', 1)]
>>> suits = 'spades hearts diamonds clubs'.split()
>>> list(itertools.product('AK', suits))  # ❷
[('A', 'spades'), ('A', 'hearts'), ('A', 'diamonds'), ('A', 'clubs'),
('K', 'spades'), ('K', 'hearts'), ('K', 'diamonds'), ('K', 'clubs')]
>>> list(itertools.product('ABC'))  # ❸
[('A',), ('B',), ('C',)]
>>> list(itertools.product('ABC', repeat=2))  # ❹
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'B'),
('B', 'C'), ('C', 'A'), ('C', 'B'), ('C', 'C')]
>>> list(itertools.product(range(2), repeat=3))
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0),
(1, 0, 1), (1, 1, 0), (1, 1, 1)]
>>> rows = itertools.product('AB', range(2), repeat=2)
>>> for row in rows: print(row)
...
('A', 0, 'A', 0)
('A', 0, 'A', 1)
('A', 0, 'B', 0)
('A', 0, 'B', 1)
('A', 1, 'A', 0)
('A', 1, 'A', 1)
('A', 1, 'B', 0)
('A', 1, 'B', 1)
('B', 0, 'A', 0)
('B', 0, 'A', 1)
('B', 0, 'B', 0)
('B', 0, 'B', 1)
('B', 1, 'A', 0)
('B', 1, 'A', 1)
('B', 1, 'B', 0)
```

```
('B', 1, 'B', 1)
```

❶ 三个字符的字符串与两个整数的值域得到的笛卡儿积是六个元组（因为 `3 * 2` 等于 6）。

❷ 两张牌（`'AK'`）与四种花色得到的笛卡儿积是八个元组。

❸ 如果传入一个可迭代的对象，`product` 函数产出的是一系列只有一个元素的元组，不是特别有用。

❹ `repeat=N` 关键字参数告诉 `product` 函数重复 *N* 次处理输入的各个可迭代对象。

有些生成器函数会从一个元素中产出多个值，扩展输入的可迭代对象，如表 14-4 所示。

表**14-4**：把输入的各个元素扩展成多个输出元素的生成器函数

模块	函数	说明
`itertools`	`combinations(it, out_len)`	把 `it` 产出的 `out_len` 个元素组合在一起，然后产出
`itertools`	`combinations_with_replacement(it, out_len)`	把 `it` 产出的 `out_len` 个元素组合在一起，然后产出，包含相同元素的组合
`itertools`	`count(start=0, step=1)`	从 `start` 开始不断产出数字，按 `step` 指定的步幅增加
`itertools`	`cycle(it)`	从 `it` 中产出各个元素，存储各个元素的副本，然后按顺序重复不断地产出各个元素
`itertools`	`permutations(it, out_len=None)`	把 `out_len` 个 `it` 产出的元素排列在一起，然后产出这些排列；`out_len`

		的默认值等于 len(list(it))
itertools	repeat(item, [times])	重复不断地产出指定的元素，除非提供 times，指定次数

itertools 模块中的 count 和 repeat 函数返回的生成器"无中生有"：这两个函数都不接受可迭代的对象作为输入。14.8.1 节见过 itertools.count 函数。cycle 生成器会备份输入的可迭代对象，然后重复产出对象中的元素。示例 14-19 演示 count、repeat 和 cycle 的用法。

示例 **14-19**　演示 count、repeat 和 cycle 的用法

```
>>> ct = itertools.count()  # ❶
>>> next(ct)  # ❷
0
>>> next(ct), next(ct), next(ct)  # ❸
(1, 2, 3)
>>> list(itertools.islice(itertools.count(1, .3), 3))  # ❹
[1, 1.3, 1.6]
>>> cy = itertools.cycle('ABC')  # ❺
>>> next(cy)
'A'
>>> list(itertools.islice(cy, 7))  # ❻
['B', 'C', 'A', 'B', 'C', 'A', 'B']
>>> rp = itertools.repeat(7)  # ❼
>>> next(rp), next(rp)
(7, 7)
>>> list(itertools.repeat(8, 4))  # ❽
[8, 8, 8, 8]
>>> list(map(operator.mul, range(11), itertools.repeat(5)))  # ❾
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

❶ 使用 count 函数构建 ct 生成器。

❷ 获取 ct 中的第一个元素。

❸ 不能使用 ct 构建列表，因为 ct 是无穷的，所以我获取接下来的 3 个元素。

❹ 如果使用 islice 或 takewhile 函数做了限制，可以从 count 生成器中构建列表。

❺ 使用 'ABC' 构建一个 cycle 生成器，然后获取第一个元素——'A'。

❻ 只有受到 islice 函数的限制，才能构建列表；这里获取接下来的 7 个元素。

❼ 构建一个 repeat 生成器，始终产出数字 7。

❽ 传入 times 参数可以限制 repeat 生成器生成的元素数量：这里会生成 4 次数字 8。

❾ repeat 函数的常见用途：为 map 函数提供固定参数，这里提供的是乘数 5。

在 itertools 模块的文档中（https://docs.python.org/3/library/itertools.html），combinations、comb 和 permutations 生成器函数，连同 product 函数，称为组合学生成器（combinatoric generator）。itertools.product 函数和其余的组合学函数有紧密的联系，如示例 14-20 所示。

示例 **14-20**　组合学生成器函数会从输入的各个元素中产出多个值

```
>>> list(itertools.combinations('ABC', 2))  # ❶
[('A', 'B'), ('A', 'C'), ('B', 'C')]
>>> list(itertools.combinations_with_replacement('ABC', 2))  # ❷
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'B'), ('B', 'C'), ('C', 'C')]
>>> list(itertools.permutations('ABC', 2))  # ❸
[('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]
>>> list(itertools.product('ABC', repeat=2))  # ❹
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'B'), ('B', 'C'),
('C', 'A'), ('C', 'B'), ('C', 'C')]
```

❶ 'ABC' 中每两个元素（len()==2）的各种组合；在生成的元组中，元素的顺序无关紧要（可以视作集合）。

❷ 'ABC' 中每两个元素（len()==2）的各种组合，包括相同元素的组

合。

❸ 'ABC' 中每两个元素（len()==2）的各种排列；在生成的元组中，元素的顺序有重要意义。

❹ 'ABC' 和 'ABC'（repeat=2 的效果）的笛卡儿积。

本节要讲的最后一组生成器函数用于产出输入的可迭代对象中的全部元素，不过会以某种方式重新排列。其中有两个函数会返回多个生成器，分别是 itertools.groupby 和 itertools.tee。这一组里的另一个生成器函数，内置的 reversed 函数，是本节所述的函数中唯一一个不接受可迭代的对象，而只接受序列为参数的函数。这在情理之中，因为 reversed 函数从后向前产出元素，而只有序列的长度已知时才能工作。不过，这个函数会按需产出各个元素，因此无需创建反转的副本。我把 itertools.product 函数划分为用于合并的生成器，列在表 14-3 中，因为那一组函数都处理多个可迭代的对象，而表 14-5 中的生成器最多只能接受一个可迭代的对象。

表**14-5**：用于重新排列元素的生成器函数

模块	函数	说明
itertools	groupby(it,key=None)	产出由两个元素组成的元素，形式为 (key, group)，其中 key 是分组标准，group 是生成器，用于产出分组里的元素
（内置）	reversed(seq)	从后向前，倒序产出 seq 中的元素；seq 必须是序列，或者是实现了 __reversed__ 特殊方法的对象
itertools	tee(it, n=2)	产出一个由 *n* 个生成器组成的元组，每个生成器用于单独产出输入的可迭代对象中的元素

示例 14-21 演示 itertools.groupby 函数和内置的 reversed 函数的用法。注意，itertools.groupby 假定输入的可迭代对象要使用分组标准排序；即使不排序，至少也要使用指定的标准分组各个元素。

示例 **14-21** `itertools.groupby` 函数的用法

```
>>> list(itertools.groupby('LLLLAAGGG'))  # ❶
[('L', <itertools._grouper object at 0x102227cc0>),
('A', <itertools._grouper object at 0x102227b38>),
('G', <itertools._grouper object at 0x102227b70>)]
>>> for char, group in itertools.groupby('LLLLAAAGG'):  # ❷
...     print(char, '->', list(group))
...
L -> ['L', 'L', 'L', 'L']
A -> ['A', 'A',]
G -> ['G', 'G', 'G']
>>> animals = ['duck', 'eagle', 'rat', 'giraffe', 'bear',
...            'bat', 'dolphin', 'shark', 'lion']
>>> animals.sort(key=len)  # ❸
>>> animals
['rat', 'bat', 'duck', 'bear', 'lion', 'eagle', 'shark',
'giraffe', 'dolphin']
>>> for length, group in itertools.groupby(animals, len):  # ❹
...     print(length, '->', list(group))
...
3 -> ['rat', 'bat']
4 -> ['duck', 'bear', 'lion']
5 -> ['eagle', 'shark']
7 -> ['giraffe', 'dolphin']
>>> for length, group in itertools.groupby(reversed(animals), len): # ❺
...     print(length, '->', list(group))
...
7 -> ['dolphin', 'giraffe']
5 -> ['shark', 'eagle']
4 -> ['lion', 'bear', 'duck']
3 -> ['bat', 'rat']
>>>
```

❶ `groupby` 函数产出 `(key, group_generator)` 这种形式的元组。

❷ 处理 `groupby` 函数返回的生成器要嵌套迭代：这里在外层使用 `for` 循环，内层使用列表推导。

❸ 为了使用 `groupby` 函数，要排序输入；这里按照单词的长度排序。

❹ 再次遍历 `key` 和 `group` 值对，把 `key` 显示出来，并把 `group` 扩展成列表。

❺ 这里使用 reverse 生成器从右向左迭代 animals。

这一组里的最后一个生成器函数是 iterator.tee，这个函数只有一个作用：从输入的一个可迭代对象中产出多个生成器，每个生成器都可以产出输入的各个元素。产出的生成器可以单独使用，如示例 14-22 所示。

示例 **14-22**　itertools.tee 函数产出多个生成器，每个生成器都可以产出输入的各个元素

```
>>> list(itertools.tee('ABC'))
[<itertools._tee object at 0x10222abc8>, <itertools._tee object at 0x10222ac0
>>> g1, g2 = itertools.tee('ABC')
>>> next(g1)
'A'
>>> next(g2)
'A'
>>> next(g2)
'B'
>>> list(g1)
['B', 'C']
>>> list(g2)
['C']
>>> list(zip(*itertools.tee('ABC')))
[('A', 'A'), ('B', 'B'), ('C', 'C')]
```

注意，这一节的示例多次把不同的生成器函数组合在一起使用。这是这些函数的优秀特性：这些函数的参数都是生成器，而返回的结果也是生成器，因此能以很多不同的方式结合在一起使用。

既然讲到了这个话题，那就介绍一下 Python 3.3 中新出现的 yield from 语句。这个语句的作用就是把不同的生成器结合在一起使用。

14.10　Python 3.3中新出现的句法：yield from

如果生成器函数需要产出另一个生成器生成的值，传统的解决方法是使用嵌套的 for 循环。

例如，下面是我们自己实现的 chain 生成器：[13]

> [13]标准库中的 itertools.chain 函数是使用 C 语言编写的。

```
>>> def chain(*iterables):
...     for it in iterables:
...         for i in it:
...             yield i
...
>>> s = 'ABC'
>>> t = tuple(range(3))
>>> list(chain(s, t))
['A', 'B', 'C', 0, 1, 2]
```

chain 生成器函数把操作依次交给接收到的各个可迭代对象处理。为此，“PEP 380 — Syntax for Delegating to a Subgenerator”（https://www.python.org/dev/peps/pep-0380/）引入了一个新句法，如下述控制台中的代码清单所示：

```
>>> def chain(*iterables):
...     for i in iterables:
...         yield from i
...
>>> list(chain(s, t))
['A', 'B', 'C', 0, 1, 2]
```

可以看出，yield from i 完全代替了内层的 for 循环。在这个示例中使用 yield from 是对的，而且代码读起来更顺畅，不过感觉更像是语法糖。除了代替循环之外，yield from 还会创建通道，把内层生成器

直接与外层生成器的客户端联系起来。把生成器当成协程使用时，这个通道特别重要，不仅能为客户端代码生成值，还能使用客户端代码提供的值。第 16 章会深入讲解协程，其中有几页会说明为什么 `yield from` 不只是语法糖而已。

一瞥 `yield from` 之后，我们回过头继续复习标准库中善于处理可迭代对象的函数。

14.11 可迭代的归约函数

表 14-6 中的函数都接受一个可迭代的对象，然后返回单个结果。这些函数叫"归约"函数、"合拢"函数或"累加"函数。其实，这里列出的每个内置函数都可以使用 `functools.reduce` 函数实现，内置是因为使用它们便于解决常见的问题。此外，对 `all` 和 `any` 函数来说，有一项重要的优化措施是 `reduce` 函数做不到的：这两个函数会短路（即一旦确定了结果就立即停止使用迭代器）。参见示例 14-23 中 `any` 函数的最后一个测试。

表**14-6**：读取迭代器，返回单个值的内置函数

模块	函数	说明
（内置）	`all(it)`	`it` 中的所有元素都为真值时返回 `True`，否则返回 `False`；`all([])` 返回 `True`
（内置）	`any(it)`	只要 `it` 中有元素为真值就返回 `True`，否则返回 `False`；`any([])` 返回 `False`
（内置）	`max(it, [key=,] [default=])`	返回 `it` 中值最大的元素；*`key` 是排序函数，与 `sorted` 函数中的一样；如果可迭代的对象为空，返回 `default`
（内置）	`min(it, [key=,] [default=])`	返回 `it` 中值最小的元素；#`key` 是排序函数，与 `sorted` 函数中的一样；如果可迭代的对象为空，返回 `default`
`functools`	`reduce(func, it, [initial])`	把前两个元素传给 `func`，然后把计算结果和第三个元素传给 `func`，以此类推，返回最后的结果；如果提供了 `initial`，把它当作第一个元素传入
（内置）	`sum(it, start=0)`	`it` 中所有元素的总和，如果提供可选的 `start`，会把它加上（计算浮点数的加法时，可以使用 `math.fsum` 函数提高精度）

* 也可以像这样调用：max(arg1, arg2, ..., [key=?])，此时返回参数中的最大值。

也可以像这样调用：min(arg1, arg2, ..., [key=?])，此时返回参数中的最小值。

all 和 any 函数的操作演示如示例 14-23 所示。

示例 **14-23**　把几个序列传给 all 和 any 函数后得到的结果

```
>>> all([1, 2, 3])
True
>>> all([1, 0, 3])
False
>>> all([])
True
>>> any([1, 2, 3])
True
>>> any([1, 0, 3])
True
>>> any([0, 0.0])
False
>>> any([])
False
>>> g = (n for n in [0, 0.0, 7, 8])
>>> any(g)
True
>>> next(g)
8
```

10.6 节更为深入地解释过 functools.reduce 函数。

还有一个内置的函数接受一个可迭代的对象，返回不同的值——sorted。reversed 是生成器函数，与此不同，sorted 会构建并返回真正的列表。毕竟，要读取输入的可迭代对象中的每一个元素才能排序，而且排序的对象是列表，因此 sorted 操作完成后返回排序后的列表。我在这里提到 sorted，是因为它可以处理任意的可迭代对象。

当然，sorted 和这些归约函数只能处理最终会停止的可迭代对象。否则，这些函数会一直收集元素，永远无法返回结果。

下面，我们回过头来分析内置的 iter() 函数，它还有一个鲜为人知的

特性没有介绍。

14.12 深入分析iter函数

如前所述，在 Python 中迭代对象 x 时会调用 iter(x)。

可是，iter 函数还有一个鲜为人知的用法：传入两个参数，使用常规的函数或任何可调用的对象创建迭代器。这样使用时，第一个参数必须是可调用的对象，用于不断调用（没有参数），产出各个值；第二个值是哨符，这是个标记值，当可调用的对象返回这个值时，触发迭代器抛出 StopIteration 异常，而不产出哨符。

下述示例展示如何使用 iter 函数掷骰子，直到掷出 1 点为止：[14]

> [14]需要在这个示例的最前面添加一句：from random import randint。——编者注

```
>>> def d6():
...     return randint(1, 6)
...
>>> d6_iter = iter(d6, 1)
>>> d6_iter
<callable_iterator object at 0x00000000029BE6A0>
>>> for roll in d6_iter:
...     print(roll)
...
4
3
6
3
```

注意，这里的 iter 函数返回一个 callable_iterator 对象。示例中的 for 循环可能运行特别长的时间，不过肯定不会打印 1，因为 1 是哨符。与常规的迭代器一样，这个示例中的 d6_iter 对象一旦耗尽就没用了。如果想重新开始，必须再次调用 iter(...)，重新构建迭代器。

内置函数 iter 的文档（https://docs.python.org/3/library/functions.html#iter）中有个实用的例子。这段代码逐行读取文件，直到遇到空行或者到达文件末尾为止：

```
with open('mydata.txt') as fp:
```

```
    for line in iter(fp.readline, '\n'):
        process_line(line)
```

结束本章之前，我要举个实用的例子，说明如何使用生成器高效处理大量数据。

14.13　案例分析：在数据库转换工具中使用生成器

几年前，我在 BIREME 工作，这是 PAHO/WHO（Pan-American Health Organization/World Health Organization，泛美卫生组织 / 世界卫生组织）在圣保罗运营的一家数字图书馆。 BIREME 制作的众多书目数据集中包含 LILACS（Latin American and Caribbean Health Sciences index，拉美和加勒比地区健康科学索引）和 SciELO（Scientific Electronic Library Online，电子科学在线图书馆），这两个数据库完整索引了这一地区发布的科学和技术作品。

从 20 世纪 80 年代后期开始，管理 LILACS 的数据库系统是 CDS/ISIS。这是 UNESCO 开发的非关系型文档数据库，后来为了在 GNU/Linux 服务器上运行，BIREME 使用 C 语言重写了。我的工作之一是探索替代方案，把 LILACS 移植到现代的开源文档数据库（最终还要移植大得多的 SciELO），例如 CouchDB 或 MongoDB。

在探索的过程中，我编写了一个 Python 脚本——isis2json.py，把 CDS/ISIS 文件转换成适合导入 CouchDB 或 MongoDB 的 JSON 文件。起初，这个脚本读取文件的是 CDS/ISIS 导出的 ISO-2709 格式。读写过程必须采用渐进方式，因为完整的数据集比主内存大得多。解决方法很简单：主 `for` 循环每次迭代时从 .iso 文件中读取一个记录，转换后将其写入 .json 文件。

然而，在实际操作中有必要让 isis2json.py 支持 CDS/ISIS 的另一种数据格式——BIREME 在生产环境中使用的二进制 .mst 文件，避免导出为 ISO-2709 格式时消耗过多资源。

现在我遇到一个问题：用来读取 ISO-2709 和 .mst 文件的库提供的 API 差别很大。而输出 JSON 格式的循环已经很复杂了，因为这个脚本要接受多个命令行选项，每次输出时调整记录的结构。在同一个 `for` 循环中使用两个不同的 API，同时还要生成 JSON，这样太难以管理了。

解决方法是隔离读取逻辑，写进两个生成器函数中：一个函数支持一种输入格式。最终，我把 isis2json.py 脚本分成了四个函数。使用 Python 2

编写的主脚本如示例 A-5，带依赖的完整源码在 GitHub 中的 fluentpython/isis2json 仓库里（https://github.com/fluentpython/isis2json）。

下面概览这个脚本的结构。

`main`

　　`main` 函数使用 `argparse` 模块读取命令行选项，用于配置输出记录的结构。根据输入文件的扩展名，`main` 函数会选择一个合适的生成器函数，逐个读取数据，然后产出记录。

`iter_iso_records`

　　这个生成器函数用于读取 .iso 文件（假设是 ISO-2709 格式），有两个参数：一个是文件名；另一个是 `isis_json_type`，即一个与记录结构有关的选项。在这个函数的 `for` 循环中，每次迭代读取一个记录，然后创建一个空字典，把数据填充进字段之后产出字典。

`iter_mst_records`

　　这也是一个生成器函数，用于读取 .mst 文件。[15] 阅读 isis2json.py 脚本的源码后你会发现，这个函数没有 `iter_iso_records` 函数简单，不过接口和整体结构是相同的：参数是文件名和 `isis_json_type`，`for` 循环每次迭代时构建并产出一个字典，表示一个记录。

> [15]用来读取复杂的 .mst 二进制文件的库其实是用 Java 编写的，因此只有使用 Jython 解释器 2.5 或以上版本执行 isis2json.py 脚本才能使用这个功能。详情参见仓库里的 README.rst 文件（https://github.com/fluentpython/isis2json/blob/master/README.rst）。因为依赖在需要使用的生成器函数中导入，所以即便只有一个外部依赖可用，这个脚本仍能运行。

`write_json`

　　这个函数把记录输出为 JSON 格式，而且一次输出一个记录。它的参数很多，其中第一个参数（`input_gen`）是对某个生成器函数的引用：`iter_iso_records` 或 `iter_mst_records`。`write_json` 函数的主 `for` 循环迭代 `input_gen` 引用的生成器产出的字典，根据命令行选项设定的方式处理，然后把 JSON 格式的记录附加到输出文件里。

我利用生成器函数解耦了读逻辑和写逻辑。当然，解耦二者最简单的方式是，把所有记录读进内存，然后写入硬盘。可是这样并不可行，因为数据集很大。而使用生成器的话，可以交叉读写，因此这个脚本可以处理任意大小的文件。

现在，如果 isis2json.py 脚本需要再支持一种输入格式，比如说美国国会图书馆用于表示 ISO-2709 格式数据的 MARCXML 文档格式，只需再添加一个生成器函数，实现读逻辑，而复杂的 `write_json` 函数无需任何改动。

这不是什么尖端科技，可是通过这个实例我们看到了生成器的灵活性。使用生成器处理数据库时，我们把记录看成数据流，这样消耗的内存量最低，而且不管数据有多大都能处理。只要管理着大型数据集，都有可能在实践中找到机会使用生成器。

下一节讨论暂时要跳过的一个生成器特性。为什么要跳过呢？原因如下。

14.14　把生成器当成协程

Python 2.2 引入了 `yield` 关键字实现的生成器函数，大约五年后，Python 2.5 实现了“PEP 342 — Coroutines via Enhanced Generators”（https://www.python.org/dev/peps/pep-0342/）。这个提案为生成器对象添加了额外的方法和功能，其中最值得关注的是 `.send()` 方法。

与 `.__next__()` 方法一样，`.send()` 方法致使生成器前进到下一个 `yield` 语句。不过，`.send()` 方法还允许使用生成器的客户把数据发给自己，即不管传给 `.send()` 方法什么参数，那个参数都会成为生成器函数定义体中对应的 `yield` 表达式的值。也就是说，`.send()` 方法允许在客户代码和生成器之间双向交换数据。而 `.__next__()` 方法只允许客户从生成器中获取数据。

这是一项重要的“改进”，甚至改变了生成器的本性：像这样使用的话，生成器就变身为协程。在 PyCon US 2009 期间举办的一场著名的课程中（http://www.dabeaz.com/coroutines/），David Beazley（可能是 Python 社区中在协程方面最多产的作者和演讲者）提醒道：

- 生成器用于生成供迭代的数据
- 协程是数据的消费者
- 为了避免脑袋炸裂，不能把这两个概念混为一谈
- 协程与迭代无关
- 注意，虽然在协程中会使用 `yield` 产出值，但这与迭代无关[16]

——David Beazley
“A Curious Course on Coroutines and Concurrency”

[16]摘自“A Curious Course on Coroutines and Concurrency”（http://www.dabeaz.com/coroutines/Coroutines.pdf）的第 33 张幻灯片，题

> 为“Keeping It Straight”。

我会遵从他的建议，至此结束本章（因为本章真正讨论的是迭代技术），而不涉及把生成器当成协程使用的 `send` 方法和其他特性。第 16 章会讨论协程。

14.15　本章小结

Python 语言对迭代的支持如此深入，因此我经常说，Python 已经融合（grok）了迭代器。[17]Python 从语义上集成迭代器模式是个很好的例证，说明设计模式在各种编程语言中使用的方式并不相同。在 Python 中，自己动手实现的典型迭代器（如示例 14-4 所示）没有实际用途，只能用作教学示例。

> [17]根据新黑客字典（Jargon file，http://catb.org/~esr/jargon/html/G/grok.html），grok 的意思不仅是学会了新知识，还要充分吸收知识，做到"人剑合一"。

本章中编写了一个类的几个版本，用于读取内容可能很多的文件，并迭代里面的单词。因为用了生成器，所以在重构的过程中，`Sentence` 类越来越简短，越来越易于阅读。最终，我们知道了生成器的工作原理。

后来，我们编写了一个用于生成等差数列的生成器，还说明了如何利用 `itertools` 模块做简化。随后，概览了标准库中 24 个通用的生成器函数。

接着，我们分析了内置的 `iter` 函数：首先说明，以 `iter(o)` 的形式调用时返回的是迭代器；之后分析，以 `iter(func, sentinel)` 的形式调用时，能使用任何函数构建迭代器。

分析实例时，我说明了一个数据库转换工具的实现方式，指明如何使用生成器函数解耦读写逻辑，如何高效处理大型数据集，以及如何轻易支持多种数据输入格式。

本章还提到了 Python 3.3 中新出现的 `yield from` 句法，还有协程。这里只对二者做了简单介绍，本书后面会更为深入地讨论。

14.16　延伸阅读

在 Python 语言参考手册中，“6.2.9. Yield expressions”（https://docs.python.org/3/reference/expressions.html#yieldexpr）从技术层面深入说明了生成器。定义生成器函数的 PEP 是“PEP 255—Simple Generators”（https://www.python.org/dev/peps/pep-0255/）。

`itertools` 模块的文档（https://docs.python.org/3/library/itertools.html）写得很棒，包含大量示例。虽然那个模块里的函数是使用 C 语言实现的，不过文档展示了如何使用 Python 实现部分函数，这通常要利用模块里的其他函数。用法示例也很好，例如，有一个代码片段说明如何使用 `accumulate` 函数计算带利息的分期还款，得出每次要还多少。文档中还有一节是“Itertools Recipes”（https://docs.python.org/3/library/itertools.html#itertools-recipes），说明如何使用 `itertools` 模块中的现有函数实现额外的高性能函数。

在 David Beazley 与 Brian K. Jones 的《Python Cookbook（第 3 版）中文版》一书中，第 4 章有 16 个诀窍涵盖了这个话题，虽然角度不同，但都关注实际应用。

“What's New in Python 3.3”（参见“PEP 380: Syntax for Delegating to a Subgenerator”，https://docs.python.org/3/whatsnew/3.3.html#pep-380-syntax-for-delegating-to-a-subgenerator）通过示例说明了 `yield from` 句法。本书 16.7 节和 16.8 节还会讨论这个句法。

如果你对文档数据库感兴趣，想进一步了解 14.13 节的背景，可以阅读我发布在 Code4Lib Journal（涵盖图书馆与技术交集）上的论文，题为“From ISIS to CouchDB: Databases and Data Models for Bibliographic Records”（http://journal.code4lib.org/articles/4893），其中有一节对 isis2json.py 脚本做了说明。这篇论文的剩余内容说明文档数据库（如 CouchDB 和 MongoDB）实现半结构化数据模型的方式，以及为什么这种模型比关系模型更适合用于收集书目数据。

杂谈

生成器函数的语法糖多一些更好

> 在设计不同目的的控制和显示设备时，设计师需要确认它们之间具有明显差异。
>
> ——Donald Norman
> 《设计心理学》

在编程语言中，源码是“控制和显示设备”。我觉得 Python 设计得特别好，源码的可读性通常很高，好像伪代码一样。可是，没有什么是完美的。Guido van Rossum 应该遵从 Donald Norman 的建议（如上述引文），引入新的关键字，用于定义生成器函数，而不该继续使用 `def`。其实，“PEP 255 — Simple Generators”（https://www.python.org/dev/peps/pep-0255/）中的“BDFL Pronouncements”一节已经提议：

> 深藏于定义体中的“yield”语句不足以提醒语义发生了重大变化。

可是，Guido 讨厌引入新关键字，而且觉得这项提议没有说服力，因此我们只好被迫接受 `def`。

沿用函数句法定义生成器会导致几个不好的后果。在 Politz 等人发布的试验成果论文“Python, the Full Monty: A Tested Semantics for the Python Programming Language”[18] 中，有个简单的生成器函数示例（这篇论文的 4.1 节）：

```
def f(): x=0
    while True:
        x += 1
        yield x
```

然后，论文的作者指出，我们无法通过函数调用抽象产出这个过程（如示例 14-24 所示）。

示例 **14-24** “（这样）似乎能简单地抽象产出这个过程”（Politz 等人）

```
def f():
    def do_yield(n):
        yield n
    x = 0
    while True:
        x += 1
        do_yield(x)
```

如果调用示例 14-24 中的 `f()`，会得到一个无限循环，而不是生成器，因为 `yield` 关键字只能把最近的外层函数变成生成器函数。虽然生成器函数看起来像函数，可是我们不能通过简单的函数调用把职责委托给另一个生成器函数。与此相比，Lua 语言就没有强加这一限制。在 Lua 中，协程可以调用其他函数，而且其中任何一个函数都能把职责交给原来的调用方。

Python 新引入的 `yield from` 句法允许生成器或协程把工作委托给第三方完成，这样就无需嵌套 `for` 循环作为变通了。在函数调用前面加上 `yield from` 能“解决”示例 14-24 中的问题，如示例 14-25 所示。

示例 **14-25**　这样才能简单地抽象产出这个过程

```
def f():
    def do_yield(n):
        yield n
    x = 0
    while True:
        x += 1
        yield from do_yield(x)
```

沿用 `def` 声明生成器犯了可用性方面的错误，而 Python 2.5 引入的协程（也写成包含 `yield` 关键字的函数）把这个问题进一步恶化了。在协程中，`yield` 碰巧（通常）出现在赋值语句的右手边，因为 `yield` 用于接收客户传给 `.send()` 方法的参数。正如 David Beazley 所说的：

> 尽管有一些相同之处，但是生成器和协程基本上是两个不同的概念。[19]

我觉得协程也应该有专用的关键字。读到后文你会发现，协程经常会用到特殊的装饰器，这样就能与其他的函数区分开。可是，生成器函数不常使用装饰器，因此我们不得不扫描函数的定义体，看有没有 `yield` 关键字，以此判断它究竟是普通的函数，还是完全不同的洪水猛兽。

也许有人会说，这么做是为了在不增加句法的前提下支持这些特性，即便添加额外的句法，也只是“语法糖”。可是，如果能让不同的特性看起来也不同，那么我更喜欢语法糖。Lisp 代码难以阅读的主要原因就是缺少语法糖，这也导致 Lisp 语言中的所有结构看起来都像是函数调用。

生成器与迭代器的语义对比

思考迭代器与生成器之间的关系时，至少可以从三方面入手。

第一方面是接口。Python 的迭代器协议定义了两个方法：`__next__` 和 `__iter__`。生成器对象实现了这两个方法，因此从这方面来看，所有生成器都是迭代器。由此可以得知，内置的 `enumerate()` 函数创建的对象是迭代器：

```
>>> from collections import abc
>>> e = enumerate('ABC')
>>> isinstance(e, abc.Iterator)
True
```

第二方面是实现方式。从这个角度来看，生成器这种 Python 语言结构可以使用两种方式编写：含有 `yield` 关键字的函数，或者生成器表达式。调用生成器函数或者执行生成器表达式得到的生成器对象属于语言内部的 `GeneratorType` 类型（https://docs.python.org/3/library/types.html#types.GeneratorType）。从这方面来看，所有生成器都是迭代器，因为 `GeneratorType` 类型的实例实现了迭代器接口。不过，我们可以编写不是生成器的迭代器，方法是实现经典的迭代器模式，如示例 14-4 所示，或者使用 C 语言编写扩展。从这方面来看，`enumerate` 对象不是生成器：

```
>>> import types
>>> e = enumerate('ABC')
>>> isinstance(e, types.GeneratorType)
False
```

这是因为 `types.GeneratorType` 类型（https://docs.python.org/3/library/types.html#types.GeneratorType）是这样定义的："生成器－迭代器对象的类型，调用生成器函数时生成。"

第三方面是概念。根据《设计模式：可复用面向对象软件的基础》一书的定义，在典型的迭代器设计模式中，迭代器用于遍历集合，从中产出元素。迭代器可能相当复杂，例如，遍历树状数据结构。但是，不管典型的迭代器中有多少逻辑，都是从现有的数据源中读取值；而且，调用 `next(it)` 时，迭代器不能修改从数据源中读取的值，只能原封不动地产出值。

而生成器可能无需遍历集合就能生成值，例如 `range` 函数。即便依附了集合，生成器不仅能产出集合中的元素，还可能会产出派生自元素的其他值。`enumerate` 函数是很好的例子。根据迭代器设计模式的原始定义，`enumerate` 函数返回的生成器不是迭代器，因为创建的是生成器产出的元组。

从概念方面来看，实现方式无关紧要。不使用 Python 生成器对象也能编写生成器。为了表明这一点，我写了一个斐波纳契数列生成器，如示例 14-26 所示。

示例 **14-26**　fibo_by_hand.py：不使用 `GeneratorType` 实例实现斐波纳契数列生成器

```
class Fibonacci:

    def __iter__(self):
        return FibonacciGenerator()


class FibonacciGenerator:

    def __init__(self):
```

```
        self.a = 0
        self.b = 1

    def __next__(self):
        result = self.a
        self.a, self.b = self.b, self.a + self.b
        return result
```

示例 14-26 虽然可行，但只是一个愚蠢的示例。符合 Python 风格的斐波纳契数列生成器如下所示：

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
```

当然，始终可以使用生成器这个语言结构履行迭代器的基本职责：遍历集合，并从中产出元素。

事实上，Python 程序员不会严格区分二者，即便在官方文档中也把生成器称作迭代器。 Python 词汇表（https://docs.python.org/dev/glossary.html#term-iterator）对迭代器下的权威定义比较笼统，涵盖了迭代器和生成器。

> 迭代器：表示数据流的对象……

建议你读一下 Python 词汇表中对迭代器的完整定义（https://docs.python.org/3/glossary.html#term-iterator）。而在生成器的定义中（https://docs.python.org/3/glossary.html#term-generator），迭代器和生成器是同义词，“生成器”指代生成器函数，以及生成器函数构建的生成器对象。因此，在 Python 社区的行话中，迭代器和生成器在一定程度上是同义词。

Python 中最简的迭代器接口

《设计模式：可复用面向对象软件的基础》一书讲解迭代器模式时，在“实现”一节中说道：[20]

> 迭代器的最小接口由 First、Next、IsDone 和 CurrentItem 操作组成。

不过，这句话有个脚注：

> 甚至可以将 Next、IsDone 和 CurrentItem 并入到一个操作中，该操作前进到下一个对象并返回这个对象，如果遍历结束，那么这个操作返回一个特定的值（例如，0）标志该迭代结束。这样我们就使这个接口变得更小了。

这与 Python 的做法接近：只用一个 `__next__` 方法完成这项工作。不过，为了表明迭代结束，这个方法没有使用哨符，因为哨符可能不小心被忽略，而是使用 `StopIteration` 异常。简单且正确，这正是 Python 之道。

> [18]Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi,"Python: The Full Monty,"SIGPLAN Not. 48, 10 (October 2013), 217-232.

> [19]"A Curious Course on Coroutines and Concurrency"（http://www.dabeaz.com/coroutines/Coroutines.pdf），第 31 张幻灯片。

> [20]《设计模式：可复用面向对象软件的基础》第 174 页。

第 15 章　上下文管理器和 else 块

> 最终，上下文管理器可能几乎与子程序（subroutine）本身一样重要。目前，我们只了解了上下文管理器的皮毛……Basic 语言有 `with` 语句，而且很多语言都有。但是，在各种语言中 `with` 语句的作用不同，而且做的都是简单的事，虽然可以避免不断使用点号查找属性，但是不会做事前准备和事后清理。不要觉得名字一样，就意味着作用也一样。`with` 语句是非常了不起的特性。[1]
>
> ——Raymond Hettinger
> 雄辩的 Python 布道者

[1]节选自 PyCon US 2013 主题演讲“What Makes Python Awesome”（http://pyvideo.org/video/1669/keynote-3）；关于 `with` 的部分从 23:00 开始，到 26:15 结束。

本章讨论其他语言中不常见的一些流程控制特性，正因如此，Python 用户往往会忽视或没有充分使用这些特性。下面要讨论的特性有：

- `with` 语句和上下文管理器
- `for`、`while` 和 `try` 语句的 `else` 子句

`with` 语句会设置一个临时的上下文，交给上下文管理器对象控制，并且负责清理上下文。这么做能避免错误并减少样板代码，因此 API 更安全，而且更易于使用。除了自动关闭文件之外，`with` 块还有很多用途。

`else` 子句与 `with` 语句完全没有关系。可是已经写到第五部分了，我找不到其他地方介绍 `else`，又不能单写只有一页内容的一章，因此就在这一章讨论了。

下面从这个较小的话题开始，进入本章的实质内容。

15.1　先做这个，再做那个：if语句之外的else块

这个语言特性不是什么秘密，但却没有得到重视：else 子句不仅能在 if 语句中使用，还能在 for、while 和 try 语句中使用。

for/else、while/else 和 try/else 的语义关系紧密，不过与 if/else 差别很大。起初，else 这个单词的意思阻碍了我对这些特性的理解，但是最终我习惯了。

else 子句的行为如下。

for

　　仅当 for 循环运行完毕时（即 for 循环没有被 break 语句中止）才运行 else 块。

while

　　仅当 while 循环因为条件为假值而退出时（即 while 循环没有被 break 语句中止）才运行 else 块。

try

　　仅当 try 块中没有异常抛出时才运行 else 块。官方文档（https://docs.python.org/3/reference/compound_stmts.html）还指出："else 子句抛出的异常不会由前面的 except 子句处理。"

在所有情况下，如果异常或者 return、break 或 continue 语句导致控制权跳到了复合语句的主块之外，else 子句也会被跳过。

> 我觉得除了 if 语句之外，其他语句选择使用 else 关键字是个错误。else 蕴含着"排他性"这层意思，例如"要么运行这个循环，要么做那件事"。可是，在循环中，else 的语义恰好相

反："运行这个循环，然后做那件事。"因此，使用 then 关键字更好。then 在 try 语句的上下文中也说得通："尝试运行这个，然后做那件事。"可是，添加新关键字属于语言的重大变化，而 Guido 唯恐避之不及。

在这些语句中使用 else 子句通常能让代码更易于阅读，而且能省去一些麻烦，不用设置控制标志或者添加额外的 if 语句。

在循环中使用 else 子句的方式如下述代码片段所示：

```
for item in my_list:
    if item.flavor == 'banana':
        break
else:
    raise ValueError('No banana flavor found!')
```

一开始，你可能觉得没必要在 try/except 块中使用 else 子句。毕竟，在下述代码片段中，只有 dangerous_call() 不抛出异常，after_call() 才会执行，对吧？

```
try:
    dangerous_call()
    after_call()
except OSError:
    log('OSError...')
```

然而，after_call() 不应该放在 try 块中。为了清晰和准确，try 块中应该只抛出预期异常的语句。因此，像下面这样写更好：

```
try:
    dangerous_call()
except OSError:
    log('OSError...')
else:
    after_call()
```

现在很明确，try 块防守的是 dangerous_call() 可能出现的错误，而

不是 `after_call()`。而且很明显，只有 `try` 块不抛出异常，才会执行 `after_call()`。

在 Python 中，`try/except` 不仅用于处理错误，还常用于控制流程。为此，Python 官方词汇表（https://docs.python.org/3/glossary.html#term-eafp）还定义了一个缩略词（口号）。

> EAFP
>
> 　　取得原谅比获得许可容易（easier to ask for forgiveness than permission）。这是一种常见的 Python 编程风格，先假定存在有效的键或属性，如果假定不成立，那么捕获异常。这种风格简单明快，特点是代码中有很多 `try` 和 `except` 语句。与其他很多语言一样（如 C 语言），这种风格的对立面是 LBYL 风格。

接下来，词汇表定义了 LBYL。

> LBYL
>
> 　　三思而后行（look before you leap）。这种编程风格在调用函数或查找属性或键之前显式测试前提条件。与 EAFP 风格相反，这种风格的特点是代码中有很多 `if` 语句。在多线程环境中，LBYL 风格可能会在“检查”和“行事”的空当引入条件竞争。例如，对 `if key in mapping: return mapping[key]` 这段代码来说，如果在测试之后，但在查找之前，另一个线程从映射中删除了那个键，那么这段代码就会失败。这个问题可以使用锁或者 EAFP 风格解决。

如果选择使用 EAFP 风格，那就要更深入地了解 `else` 子句，并在 `try/except` 语句中合理使用。

下面探讨本章的主要话题：强大的 `with` 语句。

15.2　上下文管理器和with块

上下文管理器对象存在的目的是管理 with 语句，就像迭代器的存在是为了管理 for 语句一样。

with 语句的目的是简化 try/finally 模式。这种模式用于保证一段代码运行完毕后执行某项操作，即便那段代码由于异常、return 语句或 sys.exit() 调用而中止，也会执行指定的操作。finally 子句中的代码通常用于释放重要的资源，或者还原临时变更的状态。

上下文管理器协议包含 `__enter__` 和 `__exit__` 两个方法。with 语句开始运行时，会在上下文管理器对象上调用 `__enter__` 方法。with 语句运行结束后，会在上下文管理器对象上调用 `__exit__` 方法，以此扮演 finally 子句的角色。

最常见的例子是确保关闭文件对象。使用 with 语句关闭文件的详细说明参见示例 15-1。

示例 **15-1**　演示把文件对象当成上下文管理器使用

```
>>> with open('mirror.py') as fp:  # ❶
...     src = fp.read(60)  # ❷
...
>>> len(src)
60
>>> fp  # ❸
<_io.TextIOWrapper name='mirror.py' mode='r' encoding='UTF-8'>
>>> fp.closed, fp.encoding  # ❹
(True, 'UTF-8')
>>> fp.read(60)  # ❺
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

❶ fp 绑定到打开的文件上，因为文件的 `__enter__` 方法返回 self。

❷ 从 fp 中读取一些数据。

❸ fp 变量仍然可用。[2]

> [2]与函数和模块不同，with 块没有定义新的作用域。

❹ 可以读取 fp 对象的属性。

❺ 但是不能在 fp 上执行 I/O 操作，因为在 with 块的末尾，调用 TextIOWrapper.__exit__ 方法把文件关闭了。

示例 15-1 中标注❶的那行代码道出了不易察觉但很重要的一点：执行 with 后面的表达式得到的结果是上下文管理器对象，不过，把值绑定到目标变量上（as 子句）是在上下文管理器对象上调用 __enter__ 方法的结果。

碰巧，示例 15-1 中的 open() 函数返回 TextIOWrapper 类的实例，而该实例的 __enter__ 方法返回 self。不过，__enter__ 方法除了返回上下文管理器之外，还可能返回其他对象。

不管控制流程以哪种方式退出 with 块，都会在上下文管理器对象上调用 __exit__ 方法，而不是在 __enter__ 方法返回的对象上调用。

with 语句的 as 子句是可选的。对 open 函数来说，必须加上 as 子句，以便获取文件的引用。不过，有些上下文管理器会返回 None，因为没什么有用的对象能提供给用户。

示例 15-2 使用一个精心制作的上下文管理器执行操作，以此强调上下文管理器与 __enter__ 方法返回的对象之间的区别。

示例 **15-2**　测试 LookingGlass 上下文管理器类

```
>>> from mirror import LookingGlass
>>> with LookingGlass() as what:  ❶
...      print('Alice, Kitty and Snowdrop')  ❷
...      print(what)
...
pordwonS dna yttiK ,ecilA  ❸
YKCOWREBBAJ
>>> what  ❹
'JABBERWOCKY'
```

```
>>> print('Back to normal.')  ❺
Back to normal.
```

❶ 上下文管理器是 `LookingGlass` 类的实例；Python 在上下文管理器上调用 `__enter__` 方法，把返回结果绑定到 `what` 上。

❷ 打印一个字符串，然后打印 `what` 变量的值。

❸ 打印出的内容是反向的。

❹ 现在，`with` 块已经执行完毕。可以看出，`__enter__` 方法返回的值——即存储在 `what` 变量中的值——是字符串 `'JABBERWOCKY'`。

❺ 输出不再是反向的了。

示例 15-3 是 `LookingGlass` 类的实现。

示例 **15-3**　mirror.py：`LookingGlass` 上下文管理器类的代码

```
class LookingGlass:

    def __enter__(self):  ❶
        import sys
        self.original_write = sys.stdout.write  ❷
        sys.stdout.write = self.reverse_write  ❸
        return 'JABBERWOCKY'  ❹

    def reverse_write(self, text):  ❺
        self.original_write(text[::-1])

    def __exit__(self, exc_type, exc_value, traceback):  ❻
        import sys  ❼
        sys.stdout.write = self.original_write  ❽
        if exc_type is ZeroDivisionError:  ❾
            print('Please DO NOT divide by zero!')
            return True  ❿
        ⓫
```

❶ 除了 `self` 之外，Python 调用 `__enter__` 方法时不传入其他参数。

❷ 把原来的 `sys.stdout.write` 方法保存在一个实例属性中，供后面

使用。

❸ 为 sys.stdout.write 打猴子补丁，替换成自己编写的方法。

❹ 返回 'JABBERWOCKY' 字符串，这样才有内容存入目标变量 what。

❺ 这是用于取代 sys.stdout.write 的方法，把 text 参数的内容反转，然后调用原来的实现。

❻ 如果一切正常，Python 调用 __exit__ 方法时传入的参数是 None, None, None；如果抛出了异常，这三个参数是异常数据，如下所述。

❼ 重复导入模块不会消耗很多资源，因为 Python 会缓存导入的模块。

❽ 还原成原来的 sys.stdout.write 方法。

❾ 如果有异常，而且是 ZeroDivisionError 类型，打印一个消息……

❿ ……然后返回 True，告诉解释器，异常已经处理了。

⓫ 如果 __exit__ 方法返回 None，或者 True 之外的值，with 块中的任何异常都会向上冒泡。

> 在实际使用中，如果应用程序接管了标准输出，可能会暂时把 sys.stdout 换成类似文件的其他对象，然后再切换成原来的版本。contextlib.redirect_stdout 上下文管理器（https://docs.python.org/3/library/contextlib.html#contextlib.redirect_stdou 就是这么做的：只需传入类似文件的对象，用于替代 sys.stdout。

解释器调用 __enter__ 方法时，除了隐式的 self 之外，不会传入任何参数。传给 __exit__ 方法的三个参数列举如下。

exc_type

异常类（例如 ZeroDivisionError）。

exc_value

异常实例。有时会有参数传给异常构造方法，例如错误消息，这些参数可以使用 exc_value.args 获取。

traceback

traceback 对象。[3]

> [3]在 try/finally 语句的 finally 块中调用 sys.exc_info()（https://docs.python.org/3/library/sys.html#sys.exc_info），得到的就是 __exit__ 接收的这三个参数。鉴于 with 语句是为了取代大多数 try/finally 语句，而且通常需要调用 sys.exc_info() 来判断做什么清理操作，这种行为是合理的。

上下文管理器的具体工作方式参见示例 15-4。在这个示例中，我们在 with 块之外使用 LookingGlass 类，因此可以手动调用 __enter__ 和 __exit__ 方法。

示例 **15-4**　在 with 块之外使用 LookingGlass 类

```
>>> from mirror import LookingGlass
>>> manager = LookingGlass()  ❶
>>> manager
<mirror.LookingGlass object at 0x2a578ac>
>>> monster = manager.__enter__()  ❷
>>> monster == 'JABBERWOCKY'  ❸
eurT
>>> monster
'YKCOWREBBAJ'
>>> manager
>ca875a2x0 ta tcejbo ssalGgnikooL.rorrim<
>>> manager.__exit__(None, None, None)  ❹
>>> monster
'JABBERWOCKY'
```

❶ 实例化并审查 manager 实例。

❷ 在上下文管理器上调用 __enter__() 方法，把结果存储在 monster 中。

❸ monster 的值是字符串 'JABBERWOCKY'。打印出的 True 标识符是

反向的，因为 `stdout` 的所有输出都经过 `__enter__` 方法中打补丁的 `write` 方法处理。

❹ 调用 `manager.__exit__`，还原成之前的 `stdout.write`。

上下文管理器是相当新颖的特性，Python 社区肯定还在不断寻找新的创意用法。标准库中有一些示例。

- 在 `sqlite3` 模块中用于管理事务，参见“12.6.7.3. Using the connection as a context manager”（https://docs.python.org/3/library/sqlite3.html#using-the-connection-as-a-context-manager）。[4]
- 在 `threading` 模块中用于维护锁、条件和信号，参见“17.1.10. Using locks, conditions, and semaphores in the `with` statement”（https://docs.python.org/3/library/threading.html#using-locks-conditions-and-semaphores-in-the-with-statement）。
- 为 `Decimal` 对象的算术运算设置环境，参见 `decimal.localcontext` 函数的文档（https://docs.python.org/3/library/decimal.html#decimal.localcontext）。
- 为了测试临时给对象打补丁，参见 `unittest.mock.patch` 函数的文档（https://docs.python.org/3/library/unittest.mock.html#patch）。

> [4]在 Python 3.5 文档中是“12.6.8.3”。——编者注

标准库中还有个 `contextlib` 模块，提供一些实用工具，参见下一节。

15.3　contextlib模块中的实用工具

自己定义上下文管理器类之前，先看一下 Python 标准库文档中的“29.6 contextlib — Utilities for with-statement contexts”（https://docs.python.org/3/library/contextlib.html）。除了前面提到的 redirect_stdout 函数，contextlib 模块中还有一些类和其他函数，使用范围更广。

closing

　　如果对象提供了 close() 方法，但没有实现 __enter__/__exit__ 协议，那么可以使用这个函数构建上下文管理器。

suppress

　　构建临时忽略指定异常的上下文管理器。

@contextmanager

　　这个装饰器把简单的生成器函数变成上下文管理器，这样就不用创建类去实现管理器协议了。

ContextDecorator

　　这是个基类，用于定义基于类的上下文管理器。这种上下文管理器也能用于装饰函数，在受管理的上下文中运行整个函数。

ExitStack

　　这个上下文管理器能进入多个上下文管理器。with 块结束时，ExitStack 按照后进先出的顺序调用栈中各个上下文管理器的 __exit__ 方法。如果事先不知道 with 块要进入多少个上下文管理器，可以使用这个类。例如，同时打开任意一个文件列表中的所有文件。

显然，在这些实用工具中，使用最广泛的是 `@contextmanager` 装饰器，因此要格外留心。这个装饰器也有迷惑人的一面，因为它与迭代无关，却要使用 `yield` 语句。由此可以引出协程，这是下一章的主题。

15.4　使用@contextmanager

@contextmanager 装饰器能减少创建上下文管理器的样板代码量，因为不用编写一个完整的类，定义 __enter__ 和 __exit__ 方法，而只需实现有一个 yield 语句的生成器，生成想让 __enter__ 方法返回的值。

在使用 @contextmanager 装饰的生成器中，yield 语句的作用是把函数的定义体分成两部分：yield 语句前面的所有代码在 with 块开始时（即解释器调用 __enter__ 方法时）执行， yield 语句后面的代码在 with 块结束时（即调用 __exit__ 方法时）执行。

下面举个例子。示例 15-5 使用一个生成器函数代替示例 15-3 中定义的 LookingGlass 类。

示例 **15-5**　mirror_gen.py：使用生成器实现的上下文管理器

```
import contextlib


@contextlib.contextmanager  ❶
def looking_glass():
    import sys
    original_write = sys.stdout.write  ❷

    def reverse_write(text):  ❸
        original_write(text[::-1])

    sys.stdout.write = reverse_write  ❹
    yield 'JABBERWOCKY'  ❺
    sys.stdout.write = original_write  ❻
```

❶ 应用 contextmanager 装饰器。

❷ 贮存原来的 sys.stdout.write 方法。

❸ 定义自定义的 reverse_write 函数；在闭包中可以访问

original_write。

❹ 把 sys.stdout.write 替换成 reverse_write。

❺ 产出一个值，这个值会绑定到 with 语句中 as 子句的目标变量上。执行 with 块中的代码时，这个函数会在这一点暂停。

❻ 控制权一旦跳出 with 块，继续执行 yield 语句之后的代码；这里是恢复成原来的 sys. stdout.write 方法。

示例 15-6 是使用 looking_glass 函数的例子。

示例 **15-6**　测试 looking_glass 上下文管理器函数

```
>>> from mirror_gen import looking_glass
>>> with looking_glass() as what:  ❶
...      print('Alice, Kitty and Snowdrop')
...      print(what)
...
pordwonS dna yttiK ,ecilA
YKCOWREBBAJ
>>> what
'JABBERWOCKY'
```

❶ 与示例 15-2 唯一的区别是上下文管理器的名字：LookingGlass 变成了 looking_glass。

其实，contextlib.contextmanager 装饰器会把函数包装成实现 __enter__ 和 __exit__ 方法的类。[5]

> [5]类的名称是 _GeneratorContextManager。如果想了解具体的工作方式，可以阅读 Python 3.4 发行版中 Lib/contextlib.py 文件里的源码（https://hg.python.org/cpython/file/3.4/Lib/contextlib.py#l34）。

这个类的 __enter__ 方法有如下作用。

(1) 调用生成器函数，保存生成器对象（这里把它称为 gen）。

(2) 调用 next(gen)，执行到 yield 关键字所在的位置。

(3) 返回 next(gen) 产出的值，以便把产出的值绑定到 with/as 语句中的目标变量上。

with 块终止时，__exit__ 方法会做以下几件事。

(1) 检查有没有把异常传给 exc_type；如果有，调用 gen.throw(exception)，在生成器函数定义体中包含 yield 关键字的那一行抛出异常。

(2) 否则，调用 next(gen)，继续执行生成器函数定义体中 yield 语句之后的代码。

示例 15-5 有一个严重的错误：如果在 with 块中抛出了异常，Python 解释器会将其捕获，然后在 looking_glass 函数的 yield 表达式里再次抛出。但是，那里没有处理错误的代码，因此 looking_glass 函数会中止，永远无法恢复成原来的 sys.stdout.write 方法，导致系统处于无效状态。

示例 15-7 添加了一些代码，特别用于处理 ZeroDivisionError 异常；这样，在功能上它就与示例 15-3 中基于类的实现等效了。

示例 **15-7**　mirror_gen_exc.py：基于生成器的上下文管理器，而且实现了异常处理——从外部看，行为与示例 15-3 一样

```
import contextlib


@contextlib.contextmanager
def looking_glass():
    import sys
    original_write = sys.stdout.write

    def reverse_write(text):
        original_write(text[::-1])

    sys.stdout.write = reverse_write
    msg = ''  ❶
    try:
        yield 'JABBERWOCKY'
    except ZeroDivisionError:  ❷
        msg = 'Please DO NOT divide by zero!'
```

```
    finally:
        sys.stdout.write = original_write  ❸
        if msg:
            print(msg)  ❹
```

❶ 创建一个变量，用于保存可能出现的错误消息；与示例 15-5 相比，这是第一处改动。

❷ 处理 `ZeroDivisionError` 异常，设置一个错误消息。

❸ 撤销对 `sys.stdout.write` 方法所做的猴子补丁。

❹ 如果设置了错误消息，把它打印出来。

前面说过，为了告诉解释器异常已经处理了，`__exit__` 方法会返回 `True`，此时解释器会压制异常。如果 `__exit__` 方法没有显式返回一个值，那么解释器得到的是 `None`，然后向上冒泡异常。使用 `@contextmanager` 装饰器时，默认的行为是相反的：装饰器提供的 `__exit__` 方法假定发给生成器的所有异常都得到处理了，因此应该压制异常。[6] 如果不想让 `@contextmanager` 压制异常，必须在被装饰的函数中显式重新抛出异常。[7]

[6]把异常发给生成器的方式是使用 `throw` 方法，参见 16.5 节。

[7]这样约定的原因是，创建上下文管理器时，生成器无法返回值，只能产出值。不过，现在可以返回值了，如 16.6 节所述。届时你会看到，如果在生成器中返回值，那么会抛出异常。

使用 `@contextmanager` 装饰器时，要把 `yield` 语句放在 `try/finally` 语句中（或者放在 `with` 语句中），这是无法避免的，因为我们永远不知道上下文管理器的用户会在 `with` 块中做什么。[8]

[8]这条提示直接引用 Leonardo Rochael 的评论，他是本书的技术审校之一。说得好，Leo！

除了标准库中举的例子之外，Martijn Pieters 实现的原地文件重写上下文管理器（http://www.zopatista.com/python/2013/11/26/inplace-file-

rewriting/）是 `@contextmanager` 不错的使用实例。用法如示例 15-8 所示。

示例 **15-8**　用于原地重写文件的上下文管理器

```
import csv

with inplace(csvfilename, 'r', newline='') as (infh, outfh):
    reader = csv.reader(infh)
    writer = csv.writer(outfh)

    for row in reader:
        row += ['new', 'columns']
        writer.writerow(row)
```

`inplace` 函数是个上下文管理器，为同一个文件提供了两个句柄（这个示例中的 `infh` 和 `outfh`），以便同时读写同一个文件。这比标准库中的 `fileinput.input` 函数（https://docs.python.org/3/library/fileinput.html#fileinput.input；顺便说一下，这个函数也提供了一个上下文管理器）易于使用。

如果想学习 Martijn 实现 `inplace` 的源码（列在这篇文章中：http://www.zopatista.com/python/2013/11/26/inplace-file-rewriting/），找到 `yield` 关键字，在此之前的所有代码都用于设置上下文：先创建备份文件，然后打开并产出 `__enter__` 方法返回的可读和可写文件句柄的引用。`yield` 关键字之后的 `__exit__` 处理过程把文件句柄关闭；如果什么地方出错了，那么从备份中恢复文件。

注意，在 `@contextmanager` 装饰器装饰的生成器中，`yield` 与迭代没有任何关系。在本节所举的示例中，生成器函数的作用更像是协程：执行到某一点时暂停，让客户代码运行，直到客户让协程继续做事。第 16 章会全面讨论协程。

15.5　本章小结

本章从简单的话题入手，先讨论了 `for`、`while` 和 `try` 语句的 `else` 子句。当你习惯 `else` 子句在这些语句中的奇怪意思之后，我相信 `else` 能阐明你的意图。

然后，本章讨论了上下文管理器和 `with` 语句的作用。很快我们就知道，除了自动关闭打开的文件之外，`with` 语句还有很多用途。我们自己动手实现了一个上下文管理器——含有 `__enter__`/`__exit__` 方法的 `LookingGlass` 类，说明了如何在 `__exit__` 方法中处理异常。Raymond Hettinger 在 PyCon US 2013 上所做的主题演讲传达了一个重要的观点：`with` 不仅能管理资源，还能用于去掉常规的设置和清理代码，或者在另一个过程前后执行的操作（“What Makes Python Awesome?”，第 21 张幻灯片，https://speakerdeck.com/pyconslides/pycon-keynote-python-is-awesome-by-raymond-hettinger?slide=21）。

最后，我们分析了标准库中 `contextlib` 模块里的函数。其中，`@contextmanager` 装饰器能把包含一个 `yield` 语句的简单生成器变成上下文管理器——这比定义一个至少包含两个方法的类要更简洁。我们使用 `looking_glass` 生成器函数实现了 `LookingGlass` 类，还讨论了使用 `@contextmanager` 时如何处理异常。

`@contextmanager` 装饰器优雅且实用，把三个不同的 Python 特性结合到了一起：函数装饰器、生成器和 `with` 语句。

15.6　延伸阅读

Python 语言参考手册中的“8. Compound statements”一章（https://docs.python.org/3/reference/compound_stmts.html）全面说明了 `if`、`for`、`while` 和 `try` 语句的 `else` 子句。关于 `try/except` 语句（有 `else` 子句，或者没有）是否符合 Python 风格，Raymond Hettinger 在 Stack Overflow 中对“Is it a good practice to use try-except-else in Python?”这一问题（http://stackoverflow.com/questions/16138232/is-it-a-good-practice-to-use-try-except-else-in-python）做了精彩的回答。在 Alex Martelli 写的《Python 技术手册（第 2 版）》一书中，有一章是关于异常的，那一章极好地讨论了 EAFP 风格。Alex 认为“取得原谅比获得许可容易”是由计算领域的先驱 Grace Hopper 首先提出的。

在 Python 标准库文档中，“4. Built-in Types”一章中有一节专门说明了上下文管理器的类型（https://docs.python.org/3/library/stdtypes.html#typecontextmanager）。Python 语言参考手册中还有 `__enter__/__exit__` 两个特殊方法的文档，在“3.3.8. With Statement Context Managers”一节中（https://docs.python.org/3/reference/datamodel.html#with-statement-context-managers）。上下文管理器在“PEP 343—The‘with’Statement”（https://www.python.org/dev/peps/pep-0343/）中引入。这份 PEP 不易读懂，因为大量篇幅都在讲极端情况，以及反对其他提案。这就是 PEP 的特点。

在 PyCon US 2013 的主题演讲中，Raymond Hettinger 强调，`with` 语句是“这门语言的一项迷人特性”。在这次大会上的“Transforming Code into Beautiful, Idiomatic Python”演讲中（https://speakerdeck.com/pyconslides/transforming-code-into-beautiful-idiomatic-python-by-raymond-hettinger-1?slide=34），他还展示了上下文管理器的几个有趣应用。

Jeff Preshing 写的一篇博客文章很有趣，题为“The Python `with` Statement by Example”（http://preshing.com/20110920/the-python-with-statement-by-example/），他举例说明了 `pycairo` 图形库中的上下文管理器。

Beazley 与 Jones 在他们的《Python Cookbook（第 3 版）中文版》一书中，发明了上下文管理器的独特用途。“8.3 让对象支持上下文管理协议”一节实现了一个 `LazyConnection` 类，它的实例是上下文管理器，在 `with` 块中能自动打开和关闭网络连接。“9.22 以简单的方式定义上下文管理器”一节编写了一个用于统计代码运行时间的上下文管理器，还编写了一个使用事务修改 `list` 对象的上下文管理器：在 `with` 块中创建 `list` 实例的副本，所有改动都针对那个副本；仅当 `with` 块没有抛出异常，正常执行完毕之后，才用副本替代原来的列表。这样做简单又巧妙。

杂谈

取出面包

在 PyCon US 2013 的主题演讲“What Makes Python Awesome”中（http://pyvideo.org/video/1669/keynote-3），Raymond Hettinger 说他第一次看到 `with` 语句的提案时，觉得“有点晦涩难懂”。这和我一开始的反应类似。PEP 通常难以阅读，PEP 343 尤其如此。

然后，Hettinger 告诉我们，他认识到在计算机语言的发展历程中，子程序是最重要的发明。如果有一系列操作，如 A-B-C 和 P-B-Q，那么可以把 B 拿出来，变成子程序。这就好比把三明治的馅儿取出来，这样我们就能使用金枪鱼搭配不同的面包。可是，如果我们想把面包取出来，使用小麦面包夹不同的馅儿呢？这就是 `with` 语句实现的功能。`with` 语句是子程序的补充。Hettinger 接着说道：

> `with` 语句是非常了不起的特性。我建议你在实践中深挖这个特性的用途。使用 `with` 语句或许能做意义深远的事情。`with` 语句最好的用法还未被发掘出来。我预料，如果有好的用法，其他语言以及未来的语言会借鉴这个特性。或许，你正在参与的事情几乎与子程序的发明一样意义深远。

Hettinger 承认，他夸大了 `with` 语句的作用。尽管如此，`with` 语句仍是一个十分有用的特性。他用三明治类比，道出 `with` 语句是子程序的补充；那一刻，我的脑海中浮现了许多可能性。

如果你想让任何人信服 Python 是出色的语言，一定要观看 Hettinger

的主题演讲。关于上下文管理器的部分从 23:00 开始，到 26:15 结束。不过，整个主题演讲都很精彩。

第 16 章　协程

> 如果 Python 书籍有一定的指导作用，那么（协程就是）文档最匮乏、最鲜为人知的 Python 特性，因此表面上看是最无用的特性。
>
> ——David Beazley
Python 图书作者

字典为动词“to yield”给出了两个释义：产出和让步。对于 Python 生成器中的 `yield` 来说，这两个含义都成立。`yield item` 这行代码会产出一个值，提供给 `next(...)` 的调用方；此外，还会作出让步，暂停执行生成器，让调用方继续工作，直到需要使用另一个值时再调用 `next()`。调用方会从生成器中拉取值。

从句法上看，协程与生成器类似，都是定义体中包含 `yield` 关键字的函数。可是，在协程中，`yield` 通常出现在表达式的右边（例如，`datum = yield`），可以产出值，也可以不产出——如果 `yield` 关键字后面没有表达式，那么生成器产出 `None`。协程可能会从调用方接收数据，不过调用方把数据提供给协程使用的是 `.send(datum)` 方法，而不是 `next(...)` 函数。通常，调用方会把值推送给协程。

`yield` 关键字甚至还可以不接收或传出数据。不管数据如何流动，`yield` 都是一种流程控制工具，使用它可以实现协作式多任务：协程可以把控制器让步给中心调度程序，从而激活其他的协程。

从根本上把 `yield` 视作控制流程的方式，这样就好理解协程了。

本书前面介绍的生成器函数作用不大，但是进行一系列功能改进之后，得到了 Python 协程。了解 Python 协程的进化过程有助于理解各个阶段改进的功能和复杂度。

本章首先要简单介绍生成器如何变成协程，然后再进入核心内容。本章涵盖以下话题：

- 生成器作为协程使用时的行为和状态

- 使用装饰器自动预激协程
- 调用方如何使用生成器对象的 `.close()` 和 `.throw(...)` 方法控制协程
- 协程终止时如何返回值
- `yield from` 新句法的用途和语义
- 使用案例——使用协程管理仿真系统中的并发活动

16.1　生成器如何进化成协程

协程的底层架构在“PEP 342—Coroutines via Enhanced Generators”（https://www.python.org/dev/peps/pep-0342/）中定义，并在 Python 2.5（2006 年）实现了。自此之后，`yield` 关键字可以在表达式中使用，而且生成器 API 中增加了 `.send(value)` 方法。生成器的调用方可以使用 `.send(...)` 方法发送数据，发送的数据会成为生成器函数中 `yield` 表达式的值。因此，生成器可以作为协程使用。协程是指一个过程，这个过程与调用方协作，产出由调用方提供的值。

除了 `.send(...)` 方法，PEP 342 还添加了 `.throw(...)` 和 `.close()` 方法：前者的作用是让调用方抛出异常，在生成器中处理；后者的作用是终止生成器。下一节和 16.5 节会说明这些方法。

协程最近的演进来自 Python 3.3（2012 年）实现的“PEP 380—Syntax for Delegating to a Subgenerator”（https://www.python.org/dev/peps/pep-0380/）。PEP 380 对生成器函数的句法做了两处改动，以便更好地作为协程使用。

- 现在，生成器可以返回一个值；以前，如果在生成器中给 `return` 语句提供值，会抛出 `SyntaxError` 异常。

- 新引入了 `yield from` 句法，使用它可以把复杂的生成器重构成小型的嵌套生成器，省去了之前把生成器的工作委托给子生成器所需的大量样板代码。

这两个最新的改动分别在 16.6 节和 16.7 节讨论。

按照本书的惯例，我们先从基本概念和示例入手，然后再深入越来越难以理解的特性。

16.2　用作协程的生成器的基本行为

示例 16-1 展示了协程的行为。

示例 **16-1**　可能是协程最简单的使用演示

```
>>> def simple_coroutine(): #  ❶
...     print('-> coroutine started')
...     x = yield  # ❷
...     print('-> coroutine received:', x)
...
>>> my_coro = simple_coroutine()
>>> my_coro  # ❸
<generator object simple_coroutine at 0x100c2be10>
>>> next(my_coro)  # ❹
-> coroutine started
>>> my_coro.send(42)  # ❺
-> coroutine received: 42
Traceback (most recent call last):  # ❻
  ...
StopIteration
```

❶ 协程使用生成器函数定义：定义体中有 `yield` 关键字。

❷ `yield` 在表达式中使用；如果协程只需从客户那里接收数据，那么产出的值是 `None`——这个值是隐式指定的，因为 `yield` 关键字右边没有表达式。

❸ 与创建生成器的方式一样，调用函数得到生成器对象。

❹ 首先要调用 `next(...)` 函数，因为生成器还没启动，没在 `yield` 语句处暂停，所以一开始无法发送数据。

❺ 调用这个方法后，协程定义体中的 `yield` 表达式会计算出 `42`；现在，协程会恢复，一直运行到下一个 `yield` 表达式，或者终止。

❻ 这里，控制权流动到协程定义体的末尾，导致生成器像往常一样抛出 `StopIteration` 异常。

协程可以身处四个状态中的一个。当前状态可以使用 inspect.getgeneratorstate(...) 函数确定，该函数会返回下述字符串中的一个。

'GEN_CREATED'

　　等待开始执行。

'GEN_RUNNING'

　　解释器正在执行。[1]

> [1]只有在多线程应用中才能看到这个状态。此外，生成器对象在自己身上调用 getgeneratorstate 函数也行，不过这样做没什么用。

'GEN_SUSPENDED'

　　在 yield 表达式处暂停。

'GEN_CLOSED'

　　执行结束。

因为 send 方法的参数会成为暂停的 yield 表达式的值，所以，仅当协程处于暂停状态时才能调用 send 方法，例如 my_coro.send(42)。不过，如果协程还没激活（即，状态是 'GEN_CREATED'），情况就不同了。因此，始终要调用 next(my_coro) 激活协程——也可以调用 my_coro.send(None)，效果一样。

如果创建协程对象后立即把 None 之外的值发给它，会出现下述错误：

```
>>> my_coro = simple_coroutine()
>>> my_coro.send(1729)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't send non-None value to a just-started generator
```

注意错误消息，它表述得相当清楚。

最先调用 next(my_coro) 函数这一步通常称为“预激”（prime）协程（即，让协程向前执行到第一个 yield 表达式，准备好作为活跃的协程使用）。

下面举个产出多个值的例子，以便更好地理解协程的行为，如示例 16-2 所示。

示例 **16-2**　产出两个值的协程

```
>>> def simple_coro2(a):
...     print('-> Started: a =', a)
...     b = yield a
...     print('-> Received: b =', b)
...     c = yield a + b
...     print('-> Received: c =', c)
...
>>> my_coro2 = simple_coro2(14)
>>> from inspect import getgeneratorstate
>>> getgeneratorstate(my_coro2) ❶
'GEN_CREATED'
>>> next(my_coro2) ❷
-> Started: a = 14
14
>>> getgeneratorstate(my_coro2) ❸
'GEN_SUSPENDED'
>>> my_coro2.send(28) ❹
-> Received: b = 28
42
>>> my_coro2.send(99) ❺
-> Received: c = 99
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> getgeneratorstate(my_coro2) ❻
'GEN_CLOSED'
```

❶ inspect.getgeneratorstate 函数指明，处于 GEN_CREATED 状态（即协程未启动）。

❷ 向前执行协程到第一个 yield 表达式，打印 -> Started: a = 14 消息，然后产出 a 的值，并且暂停，等待为 b 赋值。

❸ getgeneratorstate 函数指明，处于 GEN_SUSPENDED 状态（即协

程在 yield 表达式处暂停）。

❹ 把数字 28 发给暂停的协程；计算 yield 表达式，得到 28，然后把那个数绑定给 b。打印 -> Received: b = 28 消息，产出 a + b 的值（42），然后协程暂停，等待为 c 赋值。

❺ 把数字 99 发给暂停的协程；计算 yield 表达式，得到 99，然后把那个数绑定给 c。打印 -> Received: c = 99 消息，然后协程终止，导致生成器对象抛出 StopIteration 异常。

❻ getgeneratorstate 函数指明，处于 GEN_CLOSED 状态（即协程执行结束）。

关键的一点是，协程在 yield 关键字所在的位置暂停执行。前面说过，在赋值语句中，= 右边的代码在赋值之前执行。因此，对于 b = yield a 这行代码来说，等到客户端代码再激活协程时才会设定 b 的值。这种行为要花点时间才能习惯，不过一定要理解，这样才能弄懂异步编程中 yield 的作用（后文探讨）。

simple_coro2 协程的执行过程分为 3 个阶段，如图 16-1 所示。

(1) 调用 next(my_coro2)，打印第一个消息，然后执行 yield a，产出数字 14。

(2) 调用 my_coro2.send(28)，把 28 赋值给 b，打印第二个消息，然后执行 yield a + b，产出数字 42。

(3) 调用 my_coro2.send(99)，把 99 赋值给 c，打印第三个消息，协程终止。

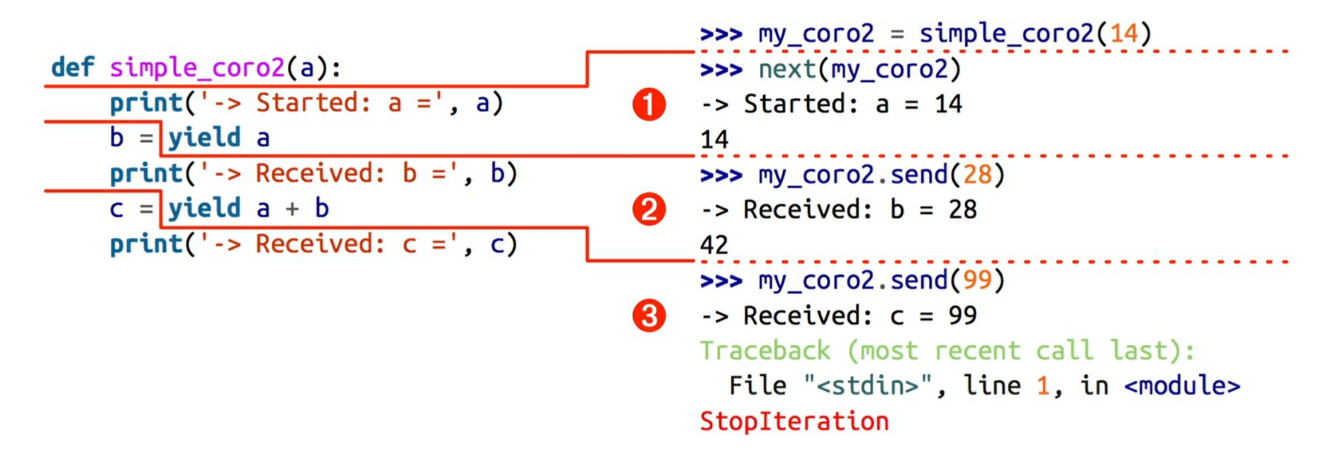


图 **16-1**：执行 **simple_coro2** 协程的 **3** 个阶段（注意，各个阶段都在 **yield** 表达式中结束，而且下一个阶段都从那一行代码开始，然后再把 **yield** 表达式的值赋给变量）

下面来看一个稍微复杂的协程示例。

16.3　示例：使用协程计算移动平均值

第 7 章讨论闭包时，我们分析了如何使用对象计算移动平均值：示例 7-8 定义的是一个简单的类；示例 7-14 定义的是一个高阶函数，用于生成一个闭包，在多次调用之间跟踪 `total` 和 `count` 变量的值。示例 16-3 展示如何使用协程实现相同的功能。[2]

> [2]这个示例的灵感来自 Jacob Holm 在 Python-ideas 邮件列表中发布的一个代码片段，他发布的消息题为“Yield-From: Finalization guarantees”（https://mail.python.org/pipermail/python-ideas/2009-April/003841.html）。在那个消息的后续回复中，那段代码有几个变体。Holm 在 003912 号消息（https://mail.python.org/pipermail/python-ideas/2009-April/003912.html）中进一步说明了自己的想法。

示例 **16-3**　coroaverager0.py：定义一个计算移动平均值的协程

```
def averager():
    total = 0.0
    count = 0
    average = None
    while True:  ❶
        term = yield average  ❷
        total += term
        count += 1
        average = total/count
```

❶ 这个无限循环表明，只要调用方不断把值发给这个协程，它就会一直接收值，然后生成结果。仅当调用方在协程上调用 `.close()` 方法，或者没有对协程的引用而被垃圾回收程序回收时，这个协程才会终止。

❷ 这里的 `yield` 表达式用于暂停执行协程，把结果发给调用方；还用于接收调用方后面发给协程的值，恢复无限循环。

使用协程的好处是，`total` 和 `count` 声明为局部变量即可，无需使用实例属性或闭包在多次调用之间保持上下文。示例 16-4 是使用 `averager` 协程的 doctest。

示例 **16-4**　coroaverager0.py：示例 16-3 中定义的移动平均值协程

的 doctest

```
>>> coro_avg = averager()  ❶
>>> next(coro_avg)  ❷
>>> coro_avg.send(10)  ❸
10.0
>>> coro_avg.send(30)
20.0
>>> coro_avg.send(5)
15.0
```

❶ 创建协程对象。

❷ 调用 `next` 函数，预激协程。

❸ 计算移动平均值：多次调用 `.send(...)` 方法，产出当前的平均值。

在上述 doctest 中（示例 16-4），调用 `next(coro_avg)` 函数后，协程会向前执行到 `yield` 表达式，产出 `average` 变量的初始值——`None`，因此不会出现在控制台中。此时，协程在 `yield` 表达式处暂停，等到调用方发送值。`coro_avg.send(10)` 那一行发送一个值，激活协程，把发送的值赋给 `term`，并更新 `total`、`count` 和 `average` 三个变量的值，然后开始 `while` 循环的下一次迭代，产出 `average` 变量的值，等待下一次为 `term` 变量赋值。

细心的读者可能迫切地想知道如何终止执行 `averager` 实例（如 `coro_avg`），因为定义体中有个无限循环。16.5 节会讨论这个话题。

讨论如何终止协程之前，我们要先谈谈如何启动协程。使用协程之前必须预激，可是这一步容易忘记。为了避免忘记，可以在协程上使用一个特殊的装饰器。接下来介绍这样一个装饰器。

16.4　预激协程的装饰器

如果不预激，那么协程没什么用。调用 my_coro.send(x) 之前，记住一定要调用 next(my_coro)。为了简化协程的用法，有时会使用一个预激装饰器。示例 16-5 中的 coroutine 装饰器是一例。[3]

> [3]网上有多个类似的装饰器。这个改自 ActiveState 中的一个诀窍——“Pipeline made of coroutines”（http://code.activestate.com/recipes/578265-pipeline-made-of-coroutines/），作者是 Chaobin Tang，而他是受到了 David Beazley 的启发。

示例 **16-5**　coroutil.py：预激协程的装饰器

```
from functools import wraps

def coroutine(func):
"""装饰器：向前执行到第一个`yield`表达式，预激`func`"""
@wraps(func)
def primer(*args,**kwargs):  ❶
    gen = func(*args,**kwargs)  ❷
    next(gen)  ❸
    return gen  ❹
return primer
```

❶ 把被装饰的生成器函数替换成这里的 primer 函数；调用 primer 函数时，返回预激后的生成器。

❷ 调用被装饰的函数，获取生成器对象。

❸ 预激生成器。

❹ 返回生成器。

示例 16-6 展示 @coroutine 装饰器的用法。请与示例 16-3 对比。

示例 **16-6**　coroaverager1.py：使用示例 16-5 中定义的 @coroutine 装饰器定义并测试计算移动平均值的协程

```
"""
用于计算移动平均值的协程

    >>> coro_avg = averager()  ❶
    >>> from inspect import getgeneratorstate
    >>> getgeneratorstate(coro_avg)  ❷
    'GEN_SUSPENDED'
    >>> coro_avg.send(10)  ❸
    10.0
    >>> coro_avg.send(30)
    20.0
    >>> coro_avg.send(5)
    15.0

"""

from coroutil import coroutine  ❹

@coroutine  ❺
def averager():  ❻
    total = 0.0
    count = 0
    average = None
    while True:
        term = yield average
        total += term
        count += 1
        average = total/count
```

❶ 调用 `averager()` 函数创建一个生成器对象，在 `coroutine` 装饰器的 `primer` 函数中已经预激了这个生成器。

❷ `getgeneratorstate` 函数指明，处于 `GEN_SUSPENDED` 状态，因此这个协程已经准备好，可以接收值了。

❸ 可以立即开始把值发给 `coro_avg`——这正是 `coroutine` 装饰器的目的。

❹ 导入 `coroutine` 装饰器。

❺ 把装饰器应用到 `averager` 函数上。

❻ 函数的定义体与示例 16-3 完全一样。

很多框架都提供了处理协程的特殊装饰器，不过不是所有装饰器都用于预激协程，有些会提供其他服务，例如勾入事件循环。比如说，异步网络库 Tornado 提供了 `tornado.gen` 装饰器（http://tornado.readthedocs.org/en/latest/gen.html）。

使用 `yield from` 句法（参见 16.7 节）调用协程时，会自动预激，因此与示例 16-5 中的 `@coroutine` 等装饰器不兼容。Python 3.4 标准库里的 `asyncio.coroutine` 装饰器（第 18 章介绍）不会预激协程，因此能兼容 `yield from` 句法。

接下来探讨协程的重要特性——用于终止协程，以及在协程中抛出异常的方法。

16.5　终止协程和异常处理

协程中未处理的异常会向上冒泡，传给 next 函数或 send 方法的调用方（即触发协程的对象）。示例 16-7 举例说明如何使用示例 16-6 中由装饰器定义的 averager 协程。

示例 **16-7**　未处理的异常会导致协程终止

```
>>> from coroaverager1 import averager
>>> coro_avg = averager()
>>> coro_avg.send(40)  # ❶
40.0
>>> coro_avg.send(50)
45.0
>>> coro_avg.send('spam')  # ❷
Traceback (most recent call last):
  ...
TypeError: unsupported operand type(s) for +=: 'float' and 'str'
>>> coro_avg.send(60)  # ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

❶ 使用 @coroutine 装饰器装饰的 averager 协程，可以立即开始发送值。

❷ 发送的值不是数字，导致协程内部有异常抛出。

❸ 由于在协程内没有处理异常，协程会终止。如果试图重新激活协程，会抛出 StopIteration 异常。

出错的原因是，发送给协程的 'spam' 值不能加到 total 变量上。

示例 16-7 暗示了终止协程的一种方式：发送某个哨符值，让协程退出。内置的 None 和 Ellipsis 等常量经常用作哨符值。Ellipsis 的优点是，数据流中不太常有这个值。我还见过有人把 StopIteration 类（类本身，而不是实例，也不抛出）作为哨符值；也就是说，是像这样使用的：my_coro.send(StopIteration)。

从 Python 2.5 开始，客户代码可以在生成器对象上调用两个方法，显式地把异常发给协程。

这两个方法是 `throw` 和 `close`。

`generator.throw(exc_type[, exc_value[, traceback]])`

　　致使生成器在暂停的 `yield` 表达式处抛出指定的异常。如果生成器处理了抛出的异常，代码会向前执行到下一个 `yield` 表达式，而产出的值会成为调用 `generator.throw` 方法得到的返回值。如果生成器没有处理抛出的异常，异常会向上冒泡，传到调用方的上下文中。

`generator.close()`

　　致使生成器在暂停的 `yield` 表达式处抛出 `GeneratorExit` 异常。如果生成器没有处理这个异常，或者抛出了 `StopIteration` 异常（通常是指运行到结尾），调用方不会报错。如果收到 `GeneratorExit` 异常，生成器一定不能产出值，否则解释器会抛出 `RuntimeError` 异常。生成器抛出的其他异常会向上冒泡，传给调用方。

> 生成器对象方法的官方文档深藏在 Python 语言参考手册中，参见“6.2.9.1.Generator-iterator methods”（https://docs.python.org/3/reference/expressions.html#generator-iterator-methods）。

下面举例说明如何使用 `close` 和 `throw` 方法控制协程。示例 16-8 列出的是接下来的例子使用的 `demo_exc_handling` 函数。

示例 **16-8**　coro_exc_demo.py：学习在协程中处理异常的测试代码

```
class DemoException(Exception):
    """为这次演示定义的异常类型。"""

def demo_exc_handling():
    print('-> coroutine started')
    while True:
        try:
            x = yield
        except DemoException:  ❶
```

```
            print('*** DemoException handled. Continuing...')
        else:  ❷
            print('-> coroutine received: {!r}'.format(x))
    raise RuntimeError('This line should never run.')  ❸
```

❶ 特别处理 DemoException 异常。

❷ 如果没有异常，那么显示接收到的值。

❸ 这一行永远不会执行。

示例 16-8 中的最后一行代码不会执行，因为只有未处理的异常才会中止那个无限循环，而一旦出现未处理的异常，协程会立即终止。

demo_exc_handling 函数的常规用法如示例 16-9 所示。

示例 **16-9**　激活和关闭 demo_exc_handling，没有异常

```
>>> exc_coro = demo_exc_handling()
>>> next(exc_coro)
-> coroutine started
>>> exc_coro.send(11)
-> coroutine received: 11
>>> exc_coro.send(22)
-> coroutine received: 22
>>> exc_coro.close()
>>> from inspect import getgeneratorstate
>>> getgeneratorstate(exc_coro)
'GEN_CLOSED'
```

如果把 DemoException 异常传入 demo_exc_handling 协程，它会处理，然后继续运行，如示例 16-10 所示。

示例 **16-10**　把 DemoException 异常传入 demo_exc_handling 不会导致协程中止

```
>>> exc_coro = demo_exc_handling()
>>> next(exc_coro)
-> coroutine started
>>> exc_coro.send(11)
-> coroutine received: 11
```

```
>>> exc_coro.throw(DemoException)
*** DemoException handled. Continuing...
>>> getgeneratorstate(exc_coro)
'GEN_SUSPENDED'
```

但是，如果传入协程的异常没有处理，协程会停止，即状态变成 `'GEN_CLOSED'`。示例 16-11 演示了这种情况。

示例 **16-11**　如果无法处理传入的异常，协程会终止

```
>>> exc_coro = demo_exc_handling()
>>> next(exc_coro)
-> coroutine started
>>> exc_coro.send(11)
-> coroutine received: 11
>>> exc_coro.throw(ZeroDivisionError)
Traceback (most recent call last):
  ...
ZeroDivisionError
>>> getgeneratorstate(exc_coro)
'GEN_CLOSED'
```

如果不管协程如何结束都想做些清理工作，要把协程定义体中相关的代码放入 `try/finally` 块中，如示例 16-12。

示例 **16-12**　coro_finally_demo.py：使用 `try/finally` 块在协程终止时执行操作

```
class DemoException(Exception):
    """为这次演示定义的异常类型。"""


def demo_finally():
    print('-> coroutine started')
    try:
        while True:
            try:
                x = yield
            except DemoException:
                print('*** DemoException handled. Continuing...')
            else:
                print('-> coroutine received: {!r}'.format(x))
    finally:
```

```
        print('-> coroutine ending')
```

Python 3.3 引入 `yield from` 结构的主要原因之一与把异常传入嵌套的协程有关。另一个原因是让协程更方便地返回值。请继续往下读，了解详情。

16.6　让协程返回值

示例 16-13 是 averager 协程的不同版本，这一版会返回结果。为了说明如何返回值，每次激活协程时不会产出移动平均值。这么做是为了强调某些协程不会产出值，而是在最后返回一个值（通常是某种累计值）。

示例 16-13 中的 averager 协程返回的结果是一个 namedtuple，两个字段分别是项数（count）和平均值（average）。我本可以只返回平均值，但是返回一个元组可以获得累积数据的另一个重要信息——项数。

示例 **16-13**　coroaverager2.py：定义一个求平均值的协程，让它返回一个结果

```
from collections import namedtuple

Result = namedtuple('Result', 'count average')


def averager():
    total = 0.0
    count = 0
    average = None
    while True:
        term = yield
        if term is None:
            break  ❶
        total += term
        count += 1
        average = total/count
    return Result(count, average)  ❷
```

❶ 为了返回值，协程必须正常终止；因此，这一版 averager 中有个条件判断，以便退出累计循环。

❷ 返回一个 namedtuple，包含 count 和 average 两个字段。在 Python 3.3 之前，如果生成器返回值，解释器会报句法错误。

下面在控制台中说明如何使用新版 averager，如示例 16-14 所示。

示例 **16-14**　coroaverager2.py：说明 averager 行为的 doctest

```
    >>> coro_avg = averager()
    >>> next(coro_avg)
    >>> coro_avg.send(10)  ❶
    >>> coro_avg.send(30)
    >>> coro_avg.send(6.5)
    >>> coro_avg.send(None)  ❷
    Traceback (most recent call last):
       ...
    StopIteration: Result(count=3, average=15.5)
```

❶ 这一版不产出值。

❷ 发送 None 会终止循环，导致协程结束，返回结果。一如既往，生成器对象会抛出 StopIteration 异常。异常对象的 value 属性保存着返回的值。

注意，return 表达式的值会偷偷传给调用方，赋值给 StopIteration 异常的一个属性。这样做有点不合常理，但是能保留生成器对象的常规行为——耗尽时抛出 StopIteration 异常。

示例 16-15 展示如何获取协程返回的值。

示例 **16-15**　捕获 StopIteration 异常，获取 averager 返回的值

```
>>> coro_avg = averager()
>>> next(coro_avg)
>>> coro_avg.send(10)
>>> coro_avg.send(30)
>>> coro_avg.send(6.5)
>>> try:
...     coro_avg.send(None)
... except StopIteration as exc:
...     result = exc.value
...
>>> result
Result(count=3, average=15.5)
```

获取协程的返回值虽然要绕个圈子，但这是 PEP 380 定义的方式，当我们意识到这一点之后就说得通了：`yield from` 结构会在内部自动捕获 `StopIteration` 异常。这种处理方式与 `for` 循环处理 `StopIteration` 异常的方式一样：循环机制使用用户易于理解的方式处理异常。对 `yield from` 结构来说，解释器不仅会捕获 `StopIteration` 异常，还会把 `value` 属性的值变成 `yield from` 表达式的值。可惜，我们无法在控制台中使用交互的方式测试这种行为，因为在函数外部使用 `yield from`（以及 `yield`）会导致句法出错。[4]

> [4]iPython 有个扩展——ipython-yf（https://github.com/tecki/ipython-yf），安装这个扩展后可以在 iPython 控制台中直接执行 `yield from`。这个扩展用于测试异步代码，可以结合 `asyncio` 模块使用。这个扩展已经提交为 Python 3.5 的补丁，但是没有被接受。参见 Python 缺陷追踪系统中的 22412 号工单： Towards an asyncio-enabled command line（http://bugs.python.org/issue22412）。

下一节会举例说明如何使用 `yield from` 结构按照 PEP 380 定义的方式获取 `averager` 协程返回的值。下面讨论 `yield from` 结构。

16.7　使用yield from

首先要知道，yield from 是全新的语言结构。它的作用比 yield 多很多，因此人们认为继续使用那个关键字多少会引起误解。在其他语言中，类似的结构使用 await 关键字，这个名称好多了，因为它传达了至关重要的一点：在生成器 gen 中使用 yield from subgen() 时，subgen 会获得控制权，把产出的值传给 gen 的调用方，即调用方可以直接控制 subgen。与此同时，gen 会阻塞，等待 subgen 终止。[5]

> [5]写作本书时，有个 PEP 正在讨论中，提议增加 await 和 async 关键字：PEP 492—Coroutines with async and await syntax（https://www.python.org/dev/peps/pep-0492/）。

第 14 章说过，yield from 可用于简化 for 循环中的 yield 表达式。例如：

```
>>> def gen():
...     for c in 'AB':
...         yield c
...     for i in range(1, 3):
...         yield i
...
>>> list(gen())
['A', 'B', 1, 2]
```

可以改写为：

```
>>> def gen():
...     yield from 'AB'
...     yield from range(1, 3)
...
>>> list(gen())
['A', 'B', 1, 2]
```

14.10 节首次提到 yield from 时举了一个例子，演示这个结构的用法，如示例 16-16 所示。[6]

> [6]示例 16-16 仅供教学使用。itertools 模块提供了优化版 chain 函数，使用 C 语言编写。

示例 **16-16** 使用 `yield from` 链接可迭代的对象

```
>>> def chain(*iterables):
...     for it in iterables:
...         yield from it
...
>>> s = 'ABC'
>>> t = tuple(range(3))
>>> list(chain(s, t))
['A', 'B', 'C', 0, 1, 2]
```

在 Beazley 与 Jones 的《Python Cookbook（第 3 版）中文版》一书中，“4.14 扁平化处理嵌套型的序列”一节有个稍微复杂（不过更有用）的 `yield from` 示例（源码在 GitHub 中，https://github.com/dabeaz/python-cookbook/blob/master/src/4/how_to_flatten_a_nested_sequence/example.py）

`yield from x` 表达式对 `x` 对象所做的第一件事是，调用 `iter(x)`，从中获取迭代器。因此，`x` 可以是任何可迭代的对象。

可是，如果 `yield from` 结构唯一的作用是替代产出值的嵌套 `for` 循环，这个结构很有可能不会添加到 Python 语言中。`yield from` 结构的本质作用无法通过简单的可迭代对象说明，而要发散思维，使用嵌套的生成器。因此，引入 `yield from` 结构的 PEP 380 才起了“Syntax for Delegating to a Subgenerator”（“把职责委托给子生成器的句法”）这个标题。

`yield from` 的主要功能是打开双向通道，把最外层的调用方与最内层的子生成器连接起来，这样二者可以直接发送和产出值，还可以直接传入异常，而不用在位于中间的协程中添加大量处理异常的样板代码。有了这个结构，协程可以通过以前不可能的方式委托职责。

若想使用 `yield from` 结构，就要大幅改动代码。为了说明需要改动的部分，PEP 380 使用了一些专门的术语。

委派生成器

　　包含 `yield from <iterable>` 表达式的生成器函数。

子生成器

从 yield from 表达式中 <iterable> 部分获取的生成器。这就是 PEP 380 的标题（“Syntax for Delegating to a Subgenerator”）中所说的“子生成器”（subgenerator）。

调用方

PEP 380 使用“调用方”这个术语指代调用委派生成器的客户端代码。在不同的语境中，我会使用“客户端”代替“调用方”，以此与委派生成器（也是调用方，因为它调用了子生成器）区分开。

> PEP 380 经常使用“迭代器”这个词指代子生成器。这样会让人误解，因为委派生成器也是迭代器。因此，我选择使用“子生成器”这个术语，与 PEP 380 的标题（“Syntax for Delegating to a Subgenerator”）保持一致。然而，子生成器可能是简单的迭代器，只实现了 __next__ 方法；但是，yield from 也能处理这种子生成器。不过，引入 yield from 结构的目的是为了支持实现了 __next__、send、close 和 throw 方法的生成器。

示例 16-17 能更好地说明 yield from 结构的用法。图 16-2 把该示例中各个相关的部分标识出来了。[7]

[7]图 16-2 的灵感来自 Paul Sokolovsky 绘制的示意图（http://flupy.org/resources/yield-from.pdf）。

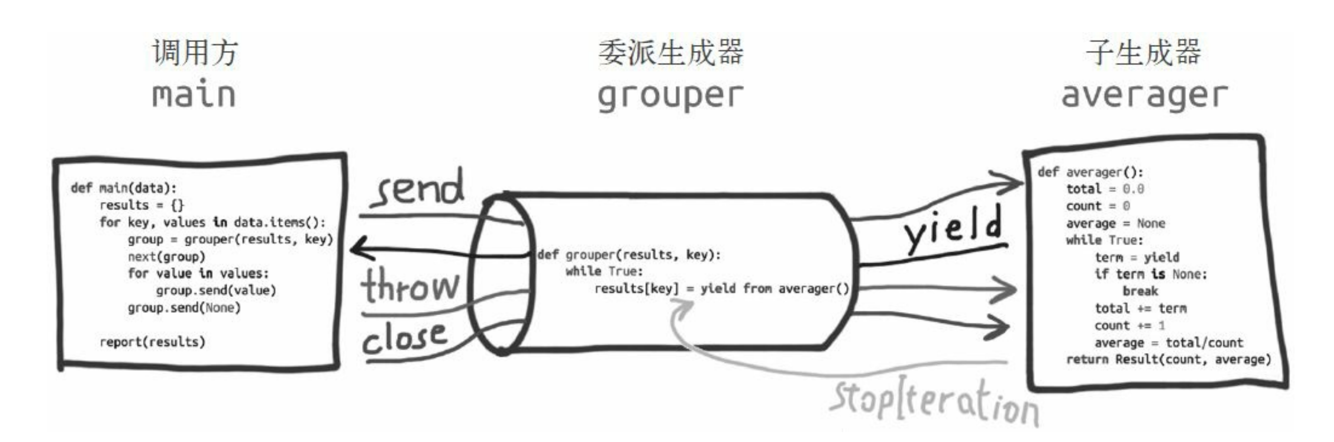


图 **16-2**：委派生成器在 **yield from** 表达式处暂停时，调用方可以直接把数据发给子生成器，子生成器再把产出的值发给调用方。子生

成器返回之后，解释器会抛出 **StopIteration** 异常，并把返回值附加到异常对象上，此时委派生成器会恢复

coroaverager3.py 脚本从一个字典中读取虚构的七年级男女学生的体重和身高。例如，'boys;m' 键对应于 9 个男学生的身高（单位是米），'girls;kg' 键对应于 10 个女学生的体重（单位是千克）。这个脚本把各组数据传给前面定义的 averager 协程，然后生成一个报告，如下所示：

```
$ python3 coroaverager3.py
 9 boys  averaging 40.42kg
 9 boys  averaging 1.39m
10 girls averaging 42.04kg
10 girls averaging 1.43m
```

示例 16-17 中列出的代码显然不是解决这个问题最简单的方案，但是通过实例说明了 yield from 结构的用法。这个示例的灵感来自“What's New in Python 3.3”一文（https://docs.python.org/3/whatsnew/3.3.html#pep-380）给出的例子。

示例 **16-17**　coroaverager3.py：使用 yield from 计算平均值并输出统计报告

```
from collections import namedtuple

Result = namedtuple('Result', 'count average')


# 子生成器
def averager():  ❶
    total = 0.0
    count = 0
    average = None
    while True:
        term = yield  ❷
        if term is None:  ❸
            break
        total += term
        count += 1
        average = total/count
        return Result(count, average)  ❹
```

```
# 委派生成器
def grouper(results, key):  ❺
    while True:  ❻
        results[key] = yield from averager()  ❼


# 客户端代码，即调用方
def main(data):  ❽
    results = {}
    for key, values in data.items():
        group = grouper(results, key)  ❾
        next(group)  ❿
        for value in values:
            group.send(value)  ⓫
        group.send(None)  # 重要!  ⓬

    # print(results)  # 如果要调试，去掉注释
    report(results)


# 输出报告
def report(results):
    for key, result in sorted(results.items()):
        group, unit = key.split(';')
        print('{:2} {:5} averaging {:.2f}{}'.format(
              result.count, group, result.average, unit))


data = {
    'girls;kg':
        [40.9, 38.5, 44.3, 42.2, 45.2, 41.7, 44.5, 38.0, 40.6, 44.5],
    'girls;m':
        [1.6, 1.51, 1.4, 1.3, 1.41, 1.39, 1.33, 1.46, 1.45, 1.43],
    'boys;kg':
        [39.0, 40.8, 43.2, 40.8, 43.1, 38.6, 41.4, 40.6, 36.3],
    'boys;m':
        [1.38, 1.5, 1.32, 1.25, 1.37, 1.48, 1.25, 1.49, 1.46],
}


if __name__ == '__main__':
    main(data)
```

❶ 与示例 16-13 中的 `averager` 协程一样。这里作为子生成器使用。

❷ main 函数中的客户代码发送的各个值绑定到这里的 term 变量上。

❸ 至关重要的终止条件。如果不这么做，使用 yield from 调用这个协程的生成器会永远阻塞。

❹ 返回的 Result 会成为 grouper 函数中 yield from 表达式的值。

❺ grouper 是委派生成器。

❻ 这个循环每次迭代时会新建一个 averager 实例；每个实例都是作为协程使用的生成器对象。

❼ grouper 发送的每个值都会经由 yield from 处理，通过管道传给 averager 实例。grouper 会在 yield from 表达式处暂停，等待 averager 实例处理客户端发来的值。averager 实例运行完毕后，返回的值绑定到 results[key] 上。while 循环会不断创建 averager 实例，处理更多的值。

❽ main 函数是客户端代码，用 PEP 380 定义的术语来说，是“调用方”。这是驱动一切的函数。

❾ group 是调用 grouper 函数得到的生成器对象，传给 grouper 函数的第一个参数是 results，用于收集结果；第二个参数是某个键。group 作为协程使用。

❿ 预激 group 协程。

⓫ 把各个 value 传给 grouper。传入的值最终到达 averager 函数中 term = yield 那一行；grouper 永远不知道传入的值是什么。

⓬ 把 None 传入 grouper，导致当前的 averager 实例终止，也让 grouper 继续运行，再创建一个 averager 实例，处理下一组值。

示例 16-17 中最后一个标号前面有个注释——“重要！”，强调这行代码（group.send(None)）至关重要：终止当前的 averager 实例，开始执行下一个。如果注释掉那一行，这个脚本不会输出任何报告。此时，把 main 函数靠近末尾的 print(results) 那行的注释去掉，你会发现，results 字典是空的。

研究为何没有收集到数据，能检验自己有没有理解 `yield from` 结构的运作方式。本书的代码仓库中有 coroaverager3.py 脚本的代码（https://github.com/fluentpython/example-code/blob/master/16-coroutine/coroaverager3.py）。原因说明如下。

下面简要说明示例 16-17 的运作方式，还会说明把 `main` 函数中调用 `group.send(None)` 那一行代码（带有“重要！”注释的那一行）去掉会发生什么事。

- 外层 `for` 循环每次迭代会新建一个 `grouper` 实例，赋值给 `group` 变量；`group` 是委派生成器。

- 调用 `next(group)`，预激委派生成器 `grouper`，此时进入 `while True` 循环，调用子生成器 `averager` 后，在 `yield from` 表达式处暂停。

- 内层 `for` 循环调用 `group.send(value)`，直接把值传给子生成器 `averager`。同时，当前的 `grouper` 实例（`group`）在 `yield from` 表达式处暂停。

- 内层循环结束后，`group` 实例依旧在 `yield from` 表达式处暂停，因此，`grouper` 函数定义体中为 `results[key]` 赋值的语句还没有执行。

- 如果外层 `for` 循环的末尾没有 `group.send(None)`，那么 `averager` 子生成器永远不会终止，委派生成器 `group` 永远不会再次激活，因此永远不会为 `results[key]` 赋值。

- 外层 `for` 循环重新迭代时会新建一个 `grouper` 实例，然后绑定到 `group` 变量上。前一个 `grouper` 实例（以及它创建的尚未终止的 `averager` 子生成器实例）被垃圾回收程序回收。

这个试验想表明的关键一点是，如果子生成器不终止，委派生成器会在 `yield from` 表达式处永远暂停。如果是这样，程序不会向前执行，因为 `yield from`（与 `yield` 一样）把控制权转交给

客户代码（即，委派生成器的调用方）了。显然，肯定有任务无法完成。

示例 16-17 展示了 `yield from` 结构最简单的用法，只有一个委派生成器和一个子生成器。因为委派生成器相当于管道，所以可以把任意数量个委派生成器连接在一起：一个委派生成器使用 `yield from` 调用一个子生成器，而那个子生成器本身也是委派生成器，使用 `yield from` 调用另一个子生成器，以此类推。最终，这个链条要以一个只使用 `yield` 表达式的简单生成器结束；不过，也能以任何可迭代的对象结束，如示例 16-16 所示。

任何 `yield from` 链条都必须由客户驱动，在最外层委派生成器上调用 `next(...)` 函数或 `.send(...)` 方法。可以隐式调用，例如使用 `for` 循环。

下面综述 PEP 380 对 `yield from` 结构的正式说明。

16.8　yield from的意义

制定 PEP 380 时，有人质疑作者 Greg Ewing 提议的语义过于复杂了。他的回应之一是："对人类来说，几乎所有最重要的信息都在靠近顶部的某个段落里。"他还引述了 PEP 380 草稿中的一段话，当时那段话是这样的：

> "把迭代器当作生成器使用，相当于把子生成器的定义体内联在 `yield from` 表达式中。此外，子生成器可以执行 `return` 语句，返回一个值，而返回的值会成为 `yield from` 表达式的值。"[8]

> [8]摘自 Python-Dev 邮件列表中的一个消息："PEP 380 (yield from a subgenerator) comments"（发布于 2009 年 3 月 21 日，https://mail.python.org/pipermail/python-dev/2009-March/087385.html）。

PEP 380 中已经没有这段宽慰人心的话，因为没有涵盖所有极端情况。不过，一开始可以这样粗略地说。

批准后的 PEP 380 在"Proposal"一节（https://www.python.org/dev/peps/pep-0380/#proposal）分六点说明了 `yield from` 的行为。这里，我几乎原封不动地引述，不过把有歧义的"迭代器"一词都换成了"子生成器"，还做了进一步说明。示例 16-17 阐明了下述四点。

- 子生成器产出的值都直接传给委派生成器的调用方（即客户端代码）。

- 使用 `send()` 方法发给委派生成器的值都直接传给子生成器。如果发送的值是 `None`，那么会调用子生成器的 `__next__()` 方法。如果发送的值不是 `None`，那么会调用子生成器的 `send()` 方法。如果调用的方法抛出 `StopIteration` 异常，那么委派生成器恢复运行。任何其他异常都会向上冒泡，传给委派生成器。

- 生成器退出时，生成器（或子生成器）中的 `return expr` 表达式会触发 `StopIteration(expr)` 异常抛出。

- `yield from` 表达式的值是子生成器终止时传给 `StopIteration`

异常的第一个参数。

yield from 结构的另外两个特性与异常和终止有关。

- 传入委派生成器的异常，除了 GeneratorExit 之外都传给子生成器的 throw() 方法。如果调用 throw() 方法时抛出 StopIteration 异常，委派生成器恢复运行。StopIteration 之外的异常会向上冒泡，传给委派生成器。

- 如果把 GeneratorExit 异常传入委派生成器，或者在委派生成器上调用 close() 方法，那么在子生成器上调用 close() 方法，如果它有的话。如果调用 close() 方法导致异常抛出，那么异常会向上冒泡，传给委派生成器；否则，委派生成器抛出 GeneratorExit 异常。

yield from 的具体语义很难理解，尤其是处理异常的那两点。Greg Ewing 做得很好，在 PEP 380 中使用英语阐述了 yield from 的语义。

Ewing 还使用伪代码（使用 Python 句法）演示了 yield from 的行为。我个人认为值得花时间研究 PEP 380 中的伪代码。不过，那段伪代码长达 40 行，看一遍很难理解。

若想研究那段伪代码，最好将其简化，只涵盖 yield from 最基本且最常见的用法。

假设 yield from 出现在委派生成器中。客户端代码驱动着委派生成器，而委派生成器驱动着子生成器。那么，为了简化涉及到的逻辑，我们假设客户端没有在委派生成器上调用 .throw(...) 或 .close() 方法。此外，我们还假设子生成器不会抛出异常，而是一直运行到终止，让解释器抛出 StopIteration 异常。

示例 16-17 中的脚本就做了这些简化逻辑的假设。其实，在真实的代码中，委派生成器应该运行到结束。下面来看一下在这个简化的美满世界中，yield from 是如何运作的。

请看示例 16-18，那里列出的代码是委派生成器的定义体中下面这一行代码的扩充：

```
RESULT = yield from EXPR
```

自己试着理解示例 16-18 中的逻辑。

示例 **16-18** 简化的伪代码，等效于委派生成器中的 `RESULT = yield from EXPR` 语句（这里针对的是最简单的情况：不支持 `.throw(...)` 和 `.close()` 方法，而且只处理 `StopIteration` 异常）

```
_i = iter(EXPR)  ❶
try:
    _y = next(_i)  ❷
except StopIteration as _e:
    _r = _e.value  ❸
else:
    while 1:  ❹
        _s = yield _y  ❺
        try:
            _y = _i.send(_s)  ❻
        except StopIteration as _e:  ❼
            _r = _e.value
            break

RESULT = _r  ❽
```

❶ `EXPR` 可以是任何可迭代的对象，因为获取迭代器 `_i`（这是子生成器）使用的是 `iter()` 函数。

❷ 预激子生成器；结果保存在 `_y` 中，作为产出的第一个值。

❸ 如果抛出 `StopIteration` 异常，获取异常对象的 `value` 属性，赋值给 `_r`——这是最简单情况下的返回值（`RESULT`）。

❹ 运行这个循环时，委派生成器会阻塞，只作为调用方和子生成器之间的通道。

❺ 产出子生成器当前产出的元素；等待调用方发送 `_s` 中保存的值。注意，这个代码清单中只有这一个 `yield` 表达式。

❻ 尝试让子生成器向前执行，转发调用方发送的 `_s`。

❼ 如果子生成器抛出 `StopIteration` 异常，获取 `value` 属性的值，赋值给 `_r`，然后退出循环，让委派生成器恢复运行。

❽ 返回的结果（`RESULT`）是 `_r`，即整个 `yield from` 表达式的值。

在这段简化的伪代码中，我保留了 PEP 380 中那段伪代码使用的变量名称。这些变量是：

`_i`（迭代器）

　　子生成器

`_y`（产出的值）

　　子生成器产出的值

`_r`（结果）

　　最终的结果（即子生成器运行结束后 `yield from` 表达式的值）

`_s`（发送的值）

　　调用方发给委派生成器的值，这个值会转发给子生成器

`_e`（异常）

　　异常对象（在这段简化的伪代码中始终是 StopIteration 实例）

除了没有处理 `.throw(...)` 和 `.close()` 方法之外，这段简化的伪代码还在子生成器上调用 `.send(...)` 方法，以此达到客户调用 `next()` 函数或 `.send(...)` 方法的目的。首次阅读时不要担心这些细微的差别。前面说过，即使 `yield from` 结构只做示例 16-18 中展示的事情，示例 16-17 也依旧能正常运行。

但是，现实情况要复杂一些，因为要处理客户对 `.throw(...)` 和 `.close()` 方法的调用，而这两个方法执行的操作必须传入子生成器。

此外，子生成器可能只是纯粹的迭代器，不支持 .throw(...) 和 .close() 方法，因此 yield from 结构的逻辑必须处理这种情况。如果子生成器实现了这两个方法，而在子生成器内部，这两个方法都会触发异常抛出，这种情况也必须由 yield from 机制处理。调用方可能会无缘无故地让子生成器自己抛出异常，实现 yield from 结构时也必须处理这种情况。最后，为了优化，如果调用方调用 next(...) 函数或 .send(None) 方法，都要转交职责，在子生成器上调用 next(...) 函数；仅当调用方发送的值不是 None 时，才使用子生成器的 .send(...) 方法。

为了方便对比，下面列出 PEP 380 中扩充 yield from 表达式的完整伪代码，而且加上了带标号的注解。示例 16-19 中的代码是一字不差复制过来的，只有标注是我自己加的。

再次说明，示例 16-19 中的代码是委派生成器的定义体中下面这一个语句的扩充：

```
RESULT = yield from EXPR
```

示例 **16-19**　伪代码，等效于委派生成器中的 RESULT = yield from EXPR 语句

```
_i = iter(EXPR)  ❶
try:
    _y = next(_i)  ❷
except StopIteration as _e:
    _r = _e.value  ❸
else:
    while 1:  ❹
        try:
            _s = yield _y  ❺
        except GeneratorExit as _e:  ❻
            try:
                _m = _i.close
            except AttributeError:
                pass
            else:
                _m()
            raise _e
        except BaseException as _e:  ❼
```

```
            _x = sys.exc_info()
            try:
                _m = _i.throw
            except AttributeError:
                raise _e
            else:  ❽
                try:
                    _y = _m(*_x)
                except StopIteration as _e:
                    _r = _e.value
                    break
        else:  ❾
            try:  ❿
                if _s is None:  ⓫
                    _y = next(_i)
                else:
                    _y = _i.send(_s)
            except StopIteration as _e:  ⓬
                _r = _e.value
                break

RESULT = _r  ⓭
```

❶ EXPR 可以是任何可迭代的对象，因为获取迭代器 _i（这是子生成器）使用的是 iter() 函数。

❷ 预激子生成器；结果保存在 _y 中，作为产出的第一个值。

❸ 如果抛出 StopIteration 异常，获取异常对象的 value 属性，赋值给 _r——这是最简单情况下的返回值（RESULT）。

❹ 运行这个循环时，委派生成器会阻塞，只作为调用方和子生成器之间的通道。

❺ 产出子生成器当前产出的元素；等待调用方发送 _s 中保存的值。这个代码清单中只有这一个 yield 表达式。

❻ 这一部分用于关闭委派生成器和子生成器。因为子生成器可以是任何可迭代的对象，所以可能没有 close 方法。

❼ 这一部分处理调用方通过 .throw(...) 方法传入的异常。同样，子

生成器可以是迭代器，从而没有 throw 方法可调用——这种情况会导致委派生成器抛出异常。

❽ 如果子生成器有 throw 方法，调用它并传入调用方发来的异常。子生成器可能会处理传入的异常（然后继续循环）；可能抛出 StopIteration 异常（从中获取结果，赋值给 _r，循环结束）；还可能不处理，而是抛出相同的或不同的异常，向上冒泡，传给委派生成器。

❾ 如果产出值时没有异常……

❿ 尝试让子生成器向前执行……

⓫ 如果调用方最后发送的值是 None，在子生成器上调用 next 函数，否则调用 send 方法。

⓬ 如果子生成器抛出 StopIteration 异常，获取 value 属性的值，赋值给 _r，然后退出循环，让委派生成器恢复运行。

⓭ 返回的结果（RESULT）是 _r，即整个 yield from 表达式的值。

这段 yield from 伪代码的大多数逻辑通过六个 try/except 块实现，而且嵌套了四层，因此有点难以阅读。此外，用到的其他流程控制关键字有一个 while、一个 if 和一个 yield。找到 while 循环、yield 表达式以及 next(...) 函数和 .send(...) 方法调用，这些代码有助于对 yield from 结构的运作方式有个整体的了解。

就在示例 16-19 所列伪代码的顶部，有行代码（标号❷）揭示了一个重要的细节：要预激子生成器。[9] 这表明，用于自动预激的装饰器（如 16.4 节定义的那个）与 yield from 结构不兼容。

> [9]Nick Coghlan 于 2009 年 4 月 5 日在 Python-ideas 邮件列表中发布的一个消息（https://mail.python.org/pipermail/python-ideas/2009-April/003954.html）中质疑，yield from 结构隐式预激是不是好主意。

在本节开头引用的那个消息中（https://mail.python.org/pipermail/python-dev/2009-March/087385.html），关于扩充 yield from 结构的伪代码，Greg Ewing 说：

> 我不是让你通过扩充的伪代码学习这个结构，那段伪代码是为了让语言专家弄明白细节。

仔细研究扩充的伪代码可能没什么用——这与你的学习方式有关。显然，分析真正使用 `yield from` 结构的代码要比深入研究实现这一结构的伪代码更有好处。不过，我见过的 `yield from` 示例几乎都使用 `asyncio` 模块做异步编程，因此要有有效的事件循环才能运行。第 18 章会多次用到 `yield from` 结构。16.11 节中有几个链接，指向使用 `yield from` 结构的一些有趣代码，而且无需事件循环。

下面分析一个使用协程的经典案例：仿真编程。这个案例没有展示 `yield from` 结构的用法，但是揭示了如何使用协程在单个线程中管理并发活动。

16.9 使用案例：使用协程做离散事件仿真

> 协程能自然地表述很多算法，例如仿真、游戏、异步 I/O，以及其他事件驱动型编程形式或协作式多任务。[10]
>
> ——Guido van Rossum 和 Phillip J. Eby
> PEP 342—Coroutines via Enhanced Generators

[10]PEP 342（https://www.python.org/dev/peps/pep-0342/）中“Motivation”一节开头的第一句话。

本节我会说明如何只使用协程和标准库中的对象实现一个特别简单的仿真系统。在计算机科学领域，仿真是协程的经典应用。第一门面向对象的语言 Simula 引入了协程这个概念，目的就是为了支持仿真。

> 下述仿真示例不是为了做学术研究。协程是 `asyncio` 包的基础构建。通过仿真系统能说明如何使用协程代替线程实现并发的活动，而且对理解第 18 章讨论的 `asyncio` 包有极大的帮助。

分析示例之前，先简单介绍一下仿真。

16.9.1 离散事件仿真简介

离散事件仿真（Discrete Event Simulation，DES）是一种把系统建模成一系列事件的仿真类型。在离散事件仿真中，仿真“钟”向前推进的量不是固定的，而是直接推进到下一个事件模型的模拟时间。假如我们抽象模拟出租车的运营过程，其中一个事件是乘客上车，下一个事件则是乘客下车。不管乘客坐了 5 分钟还是 50 分钟，一旦乘客下车，仿真钟就会更新，指向此次运营的结束时间。使用离散事件仿真可以在不到一秒钟的时间内模拟一年的出租车运营过程。这与连续仿真不同，连续仿真的仿真钟以固定的量（通常很小）不断向前推进。

显然，回合制游戏就是离散事件仿真的例子：游戏的状态只在玩家操作时变化，而且一旦玩家决定下一步怎么走了，仿真钟就会冻结。而实时

游戏则是连续仿真，仿真钟一直在运行，游戏的状态在一秒钟之内更新很多次，因此反应慢的玩家特别吃亏。

这两种仿真类型都能使用多线程或在单个线程中使用面向事件的编程技术（例如事件循环驱动的回调或协程）实现。可以说，为了实现连续仿真，在多个线程中处理实时并行的操作更自然。而协程恰好为实现离散事件仿真提供了合理的抽象。SimPy[11] 是一个实现离散事件仿真的 Python 包，通过一个协程表示离散事件仿真系统中的各个进程。

> [11]参见 SimPy 的官方文档（https://simpy.readthedocs.org/en/latest/）。不要和著名的 SymPy（http://www.sympy.org）混淆了。SymPy 是一个符号数学库，与 DES 无关。

> 在仿真领域，进程这个术语指代模型中某个实体的活动，与操作系统中的进程无关。仿真系统中的一个进程可以使用操作系统中的一个进程实现，但是通常会使用一个线程或一个协程实现。

如果对仿真感兴趣，值得研究一下 SimPy。不过，在这一节我会说明如何只使用标准库提供的功能实现一个特别简单的离散事件仿真系统。我的目的是增进你对使用协程管理并发操作的感性认知。若想理解下一节所讲的内容，要仔细研究，不过这一付出能得到很大回报，让我们洞悉 `asyncio`、Twisted 和 Tornado 等库是如何在单个线程中管理多个并发活动的。

16.9.2　出租车队运营仿真

仿真程序 taxi_sim.py 会创建几辆出租车，每辆车会拉几个乘客，然后回家。出租车首先驶离车库，四处徘徊，寻找乘客；拉到乘客后，行程开始；乘客下车后，继续四处徘徊。

四处徘徊和行程所用的时间使用指数分布生成。为了让显示的信息更加整洁，时间使用取整的分钟数，不过这个仿真程序也能使用浮点数表示耗时。[12] 每辆出租车每次的状态变化都是一个事件。图 16-3 是运行这个程序的输出示例。

> [12]我不是运营出租车队的行家，因此别太在意显示的时间。离散事件仿真经常使用指数分布。你会看到一些非常短的行程，你就假设那是一个雨天，一些乘客坐出租车只走了一个街区。在理想的城市中，即使下雨也有出租车。

```
$ python3 taxi_sim.py -s 3
taxi: 0  Event(time=0, proc=0, action='leave garage')
taxi: 0  Event(time=2, proc=0, action='pick up passenger')
taxi: 1     Event(time=5, proc=1, action='leave garage')
taxi: 1     Event(time=8, proc=1, action='pick up passenger')
taxi: 2        Event(time=10, proc=2, action='leave garage')
taxi: 2        Event(time=15, proc=2, action='pick up passenger')
taxi: 2        Event(time=17, proc=2, action='drop off passenger')
taxi: 0  Event(time=18, proc=0, action='drop off passenger')
taxi: 2        Event(time=18, proc=2, action='pick up passenger')
taxi: 2        Event(time=25, proc=2, action='drop off passenger')
taxi: 1     Event(time=27, proc=1, action='drop off passenger')
taxi: 2        Event(time=27, proc=2, action='pick up passenger')
taxi: 0  Event(time=28, proc=0, action='pick up passenger')
taxi: 2        Event(time=40, proc=2, action='drop off passenger')
taxi: 2        Event(time=44, proc=2, action='pick up passenger')
taxi: 1     Event(time=55, proc=1, action='pick up passenger')
taxi: 1     Event(time=59, proc=1, action='drop off passenger')
taxi: 0  Event(time=65, proc=0, action='drop off passenger')
taxi: 1     Event(time=65, proc=1, action='pick up passenger')
taxi: 2        Event(time=65, proc=2, action='drop off passenger')
taxi: 2        Event(time=72, proc=2, action='pick up passenger')
taxi: 0  Event(time=76, proc=0, action='going home')
taxi: 1     Event(time=80, proc=1, action='drop off passenger')
taxi: 1     Event(time=88, proc=1, action='pick up passenger')
taxi: 2        Event(time=95, proc=2, action='drop off passenger')
taxi: 2        Event(time=97, proc=2, action='pick up passenger')
taxi: 2        Event(time=98, proc=2, action='drop off passenger')
taxi: 1     Event(time=106, proc=1, action='drop off passenger')
taxi: 2        Event(time=109, proc=2, action='going home')
taxi: 1     Event(time=110, proc=1, action='going home')
*** end of events ***
```

图 **16-3**：运行 **taxi_sim.py** 创建 **3** 辆出租车的输出示例。**-s 3** 参数设置随机数生成器的种子，这样在调试和演示时可以重复运行程序，输出相同的结果。不同颜色的箭头表示不同出租车的行程 [13]

> [13]图 16-3 的彩色图片可从本书页面（http://www.ituring.com.cn/book/1564）的“随书下载”部分获取。——编者注

图 16-3 中最值得注意的一件事是，3 辆出租车的行程是交叉进行的。那些箭头是我加上的，为的是让你看清各辆出租车的行程：箭头从乘客上

车时开始，到乘客下车后结束。有了箭头，能直观地看出如何使用协程管理并发的活动。

图 16-3 中还有几件事值得注意。

- 出租车每隔 5 分钟从车库中出发。
- 0 号出租车 2 分钟后拉到乘客（`time=2`），1 号出租车 3 分钟后拉到乘客（`time=8`），2 号出租车 5 分钟后拉到乘客（`time=15`）。
- 0 号出租车拉了两个乘客（紫色箭头）：第一个乘客从 `time=2` 时上车，到 `time=18` 时下车；第二个乘客从 `time=28` 时上车，到 `time=65` 时下车——这是此次仿真中最长的行程。
- 1 号出租车拉了四个乘客（绿色箭头），在 `time=110` 时回家。
- 2 号出租车拉了六个乘客（红色箭头），在 `time=109` 时回家。这辆车最后一次行程从 `time=97` 时开始，只持续了一分钟。[14]
- 1 号出租车的第一次行程从 `time=8` 时开始，在这个过程中 2 号出租车离开了车库（`time=10`），而且完成了两次行程（那两个短的红色箭头）。
- 在此次运行示例中，所有排定的事件都在默认的仿真时间内（180 分钟）完成；最后一次事件发生在 `time=110` 时。

[14]乘客是我，我发现忘了带钱包。

仿真结束时可能还有未完成的事件。如果是这种情况，最后一条消息会是下面这样：

```
*** end of simulation time: 3 events pending ***
```

taxi_sim.py 脚本的完整代码在示例 A-6 中，本章只会列出与协程相关的部分。真正重要的函数只有两个：`taxi_process`（一个协程），以及执行仿真主循环的 `Simulator.run` 方法。

示例 16-20 是 taxi_process 函数的代码。这个协程用到了别处定义的两个对象：compute_delay 函数，返回单位为分钟的时间间隔；Event 类，一个 namedtuple，定义方式如下：

```
Event = collections.namedtuple('Event', 'time proc action')
```

在 Event 实例中，time 字段是事件发生时的仿真时间，proc 字段是出租车进程实例的编号，action 字段是描述活动的字符串。

下面逐行分析示例 16-20 中的 taxi_process 函数。

示例 **16-20**　taxi_sim.py：taxi_process 协程，实现各辆出租车的活动

```
def taxi_process(ident, trips, start_time=0):  ❶
    """每次改变状态时创建事件，把控制权让给仿真器"""
    time = yield Event(start_time, ident, 'leave garage')  ❷
    for i in range(trips):  ❸
        time = yield Event(time, ident, 'pick up passenger')  ❹
        time = yield Event(time, ident, 'drop off passenger')  ❺

    yield Event(time, ident, 'going home')  ❻
    # 出租车进程结束  ❼
```

❶ 每辆出租车调用一次 taxi_process 函数，创建一个生成器对象，表示各辆出租车的运营过程。ident 是出租车的编号（如上述运行示例中的 0、1、2）；trips 是出租车回家之前的行程数量；start_time 是出租车离开车库的时间。

❷ 产出的第一个 Event 是 'leave garage'。执行到这一行时，协程会暂停，让仿真主循环着手处理排定的下一个事件。需要重新激活这个进程时，主循环会发送（使用 send 方法）当前的仿真时间，赋值给 time。

❸ 每次行程都会执行一遍这个代码块。

❹ 产出一个 Event 实例，表示拉到乘客了。协程在这里暂停。需要重

新激活这个协程时，主循环会发送（使用 send 方法）当前的时间。

❺ 产出一个 Event 实例，表示乘客下车了。协程在这里暂停，等待主循环发送时间，然后重新激活。

❻ 指定的行程数量完成后，for 循环结束，最后产出 'going home' 事件。此时，协程最后一次暂停。仿真主循环发送时间后，协程重新激活；不过，这里没有把产出的值赋值给变量，因为用不到了。

❼ 协程执行到最后时，生成器对象抛出 StopIteration 异常。

你可以在 Python 控制台中调用 taxi_process 函数，自己“驾驶”（drive）一辆出租车[15]，如示例 16-21 所示。

> [15]描述协程的操作时经常使用“drive”这个动词，例如：客户代码把值发给协程，驱动协程。在示例 16-21 中，客户代码是你在控制台中输入的代码。（drive 一词有不同的含义，因此在不同的语境中有不同的译法，例如这个脚注所在的那句话中译为“驾驶”。——译者注）

示例 **16-21**　驱动 taxi_process 协程

```
>>> from taxi_sim import taxi_process
>>> taxi = taxi_process(ident=13, trips=2, start_time=0) ❶
>>> next(taxi) ❷
Event(time=0, proc=13, action='leave garage')
>>> taxi.send(_.time + 7) ❸
Event(time=7, proc=13, action='pick up passenger') ❹
>>> taxi.send(_.time + 23) ❺
Event(time=30, proc=13, action='drop off passenger')
>>> taxi.send(_.time + 5) ❻
Event(time=35, proc=13, action='pick up passenger')
>>> taxi.send(_.time + 48) ❼
Event(time=83, proc=13, action='drop off passenger')
>>> taxi.send(_.time + 1)
Event(time=84, proc=13, action='going home') ❽
>>> taxi.send(_.time + 10) ❾
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

❶ 创建一个生成器对象，表示一辆出租车。这辆出租车的编号是 13（ident=13），从 t=0 时开始工作，有两次行程。

❷ 预激协程；产出第一个事件。

❸ 现在可以发送当前时间。在控制台中，`_` 变量绑定的是前一个结果；这里我在时间上加 `7`，意思是这辆出租车 7 分钟后找到第一个乘客。

❹ 这个事件由 `for` 循环在第一个行程的开头产出。

❺ 发送 `_.time + 23`，表示第一个乘客的行程持续了 23 分钟。

❻ 然后，这辆出租车会徘徊 5 分钟。

❼ 最后一次行程持续 48 分钟。

❽ 两次行程完成后，`for` 循环结束，产出 `'going home'` 事件。

❾ 如果尝试再把值发给协程，会执行到协程的末尾。协程返回后，解释器会抛出 `StopIteration` 异常。

注意，在示例 16-21 中，我使用控制台模拟仿真主循环。我从 `taxi` 协程产出的 `Event` 实例中获取 `.time` 属性，随意与一个数相加，然后调用 `taxi.send` 方法发送两数之和，重新激活协程。在这个仿真系统中，各个出租车协程由 `Simulator.run` 方法中的主循环驱动。仿真“钟”保存在 `sim_time` 变量中，每次产出事件时都会更新仿真钟。

为了实例化 `Simulator` 类，taxi_sim.py 脚本的 `main` 函数构建了一个 `taxis` 字典，如下所示：

```
taxis = {i: taxi_process(i, (i + 1) * 2, i * DEPARTURE_INTERVAL)
         for i in range(num_taxis)}
sim = Simulator(taxis)
```

`DEPARTURE_INTERVAL` 的值是 `5`；如果 `num_taxis` 的值与前面的运行示例一样也是 `3`，这三行代码的作用与下述代码一样：

```
taxis = {0: taxi_process(ident=0, trips=2, start_time=0),
         1: taxi_process(ident=1, trips=4, start_time=5),
         2: taxi_process(ident=2, trips=6, start_time=10)}
sim = Simulator(taxis)
```

因此，taxis 字典的值是三个参数不同的生成器对象。例如，1 号出租车从 start_time=5 时开始，寻找四个乘客。构建 Simulator 实例只需这个字典参数。

Simulator.__init__ 方法如示例 16-22 所示。Simulator 类的主要数据结构如下。

self.events

PriorityQueue 对象，保存 Event 实例。元素可以放进（使用 put 方法）PriorityQueue 对象中，然后按 item[0]（即 Event 对象的 time 属性）依序取出（使用 get 方法）。

self.procs

一个字典，把出租车的编号映射到仿真过程中激活的进程（表示出租车的生成器对象）。这个属性会绑定前面所示的 taxis 字典副本。

示例 **16-22** taxi_sim.py：Simulator 类的初始化方法

```
class Simulator:

    def __init__(self, procs_map):
        self.events = queue.PriorityQueue() ❶
        self.procs = dict(procs_map) ❷
```

❶ 保存排定事件的 PriorityQueue 对象，按时间正向排序。

❷ 获取的 procs_map 参数是一个字典（或其他映射），可是又从中构建一个字典，创建本地副本，因为在仿真过程中，出租车回家后会从 self.procs 属性中移除，而我们不想修改用户传入的对象。

优先队列是离散事件仿真系统的基础构件：创建事件的顺序不定，放入这种队列之后，可以按照各个事件排定的时间顺序取出。例如，可能会把下面两个事件放入优先队列：

```
Event(time=14, proc=0, action='pick up passenger')
Event(time=11, proc=1, action='pick up passenger')
```

这两个事件的意思是，0 号出租车 14 分钟后拉到第一个乘客，而 1 号出租车（`time=10` 时出发）1 分钟后（`time=11`）拉到乘客。如果这两个事件在队列中，主循环从优先队列中获取的第一个事件将是 `Event(time=11, proc=1, action='pick up passenger')`。

下面分析这个仿真系统的主算法——`Simulator.run` 方法。在 `main` 函数中，实例化 `Simulator` 类之后立即就调用了这个方法，如下所示：

```
sim = Simulator(taxis)
sim.run(end_time)
```

`Simulator` 类带有注解的代码清单在示例 16-23 中，下面先概述 `Simulator.run` 方法实现的算法。

(1) 迭代表示各辆出租车的进程。

  a. 在各辆出租车上调用 `next()` 函数，预激协程。这样会产出各辆出租车的第一个事件。

  b. 把各个事件放入 `Simulator` 类的 `self.events` 属性（队列）中。

(2) 满足 `sim_time < end_time` 条件时，运行仿真系统的主循环。

  a. 检查 `self.events` 属性是否为空；如果为空，跳出循环。

  b. 从 `self.events` 中获取当前事件（`current_event`），即 `PriorityQueue` 对象中时间值最小的 `Event` 对象。

  c. 显示获取的 `Event` 对象。

  d.获取 `current_event` 的 `time` 属性，更新仿真时间。

e.把时间发给 current_event 的 proc 属性标识的协程，产出下一个事件（next_event）。

f.把 next_event 添加到 self.events 队列中，排定 next_event。

Simulator 类完整的代码如示例 16-23 所示。

示例 **16-23**　taxi_sim.py：Simulator，一个简单的离散事件仿真类；关注的重点是 run 方法

```
class Simulator:

    def __init__(self, procs_map):
        self.events = queue.PriorityQueue()
        self.procs = dict(procs_map)

    def run(self, end_time): ❶
        """排定并显示事件，直到时间结束"""
        # 排定各辆出租车的第一个事件
        for _, proc in sorted(self.procs.items()):  ❷
            first_event = next(proc)  ❸
            self.events.put(first_event)  ❹

        # 这个仿真系统的主循环
        sim_time = 0  ❺
        while sim_time < end_time:  ❻
            if self.events.empty():  ❼
                print('*** end of events ***')
                break

            current_event = self.events.get()  ❽
            sim_time, proc_id, previous_action = current_event  ❾
            print('taxi:', proc_id, proc_id * '   ', current_event)  ❿
            active_proc = self.procs[proc_id]  ⓫
            next_time = sim_time + compute_duration(previous_action)  ⓬
            try:
                next_event = active_proc.send(next_time)  ⓭
            except StopIteration:
                del self.procs[proc_id]  ⓮
            else:
                self.events.put(next_event)  ⓯
        else:  ⓰
            msg = '*** end of simulation time: {} events pending ***'
```

```
        print(msg.format(self.events.qsize()))
```

❶ run 方法只需要仿真结束时间（end_time）这一个参数。

❷ 使用 sorted 函数获取 self.procs 中按键排序的元素；用不到键，因此赋值给 _。

❸ 调用 next(proc) 预激各个协程，向前执行到第一个 yield 表达式，做好接收数据的准备。产出一个 Event 对象。

❹ 把各个事件添加到 self.events 属性表示的 PriorityQueue 对象中。如示例 16-20 中的运行示例，各辆出租车的第一个事件是 'leave garage'。

❺ 把 sim_time 变量（仿真钟）归零。

❻ 这个仿真系统的主循环：sim_time 小于 end_time 时运行。

❼ 如果队列中没有未完成的事件，退出主循环。

❽ 获取优先队列中 time 属性最小的 Event 对象；这是当前事件（current_event）。

❾ 拆包 Event 对象中的数据。这一行代码会更新仿真钟 sim_time，对应于事件发生时的时间。[16]

> [16]这通常是离散事件仿真：每次循环时仿真钟不会以固定的量推进，而是根据各个事件持续的时间推进。

❿ 显示 Event 对象，指明是哪辆出租车，并根据出租车的编号缩进。

⓫ 从 self.procs 字典中获取表示当前活动的出租车的协程。

⓬ 调用 compute_duration(...) 函数，传入前一个动作（例如，'pick up passenger'、'drop off passenger' 等），把结果加到 sim_time 上，计算出下一次活动的时间。

⓭ 把计算得到的时间发给出租车协程。协程会产出下一个事件（next_event），或者抛出 StopIteration 异常（完成时）。

⓮ 如果抛出了 StopIteration 异常，从 self.procs 字典中删除那个协程。

⓯ 否则，把 next_event 放入队列中。

⓰ 如果循环由于仿真时间到了而退出，显示待完成的事件数量（有时可能碰巧是零）。

注意，示例 16-23 中的 Simulator.run 方法有两处用到了第 15 章介绍的 else 块，而且都不在 if 语句中。

- 主 while 循环有一个 else 语句，报告仿真系统由于到达结束时间而结束，而不是由于没有事件要处理而结束。

- 靠近主 while 循环底部那个 try 语句把 next_time 发给当前的出租车进程，尝试获取下一个事件（next_event），如果成功，执行 else 块，把 next_event 放入 self.events 队列中。

我觉得，如果没有这两个 else 块，Simulator.run 方法的代码会有点难以阅读。

这个示例的要旨是说明如何在一个主循环中处理事件，以及如何通过发送数据驱动协程。这是 asyncio 包底层的基本思想，我们在第 18 章会学习这个包。

16.10　本章小结

Guido van Rossum 写道，生成器有三种不同的代码编写风格：

> 有传统的“拉取式”（迭代器）、“推送式”（例如计算平均值那个示例），还有“任务式”（读过 Dave Beazley 写的协程教程了吗……）。[17]

> [17]摘自对 Python-ideas 邮件列表中“Yield-From: Finalization guarantees”消息的回复（https://mail.python.org/pipermail/python-ideas/2009-April/003884.html）。Guido 所说的 David Beazley 写的教程是“A Curious Course on Coroutines and Concurrency”（http://www.dabeaz.com/coroutines/）。

第 14 章专门介绍了迭代器，本章则介绍了“推送式”协程，还介绍了特别简单的“任务式”——仿真示例中的出租车进程。第 18 章会在并发编程中使用这两种技术实现异步任务。

计算移动平均值的示例展示了协程的常见用途：累加器，处理接收到的值。我们知道，可以在协程上应用装饰器，预激协程；在某些情况下，这么做更方便。不过要记住，预激装饰器与协程的某些用法不兼容。尤其是 `yield from subgenerator()`，这个结构假定 `subgenerator` 没有预激，然后自动预激。

每次调用 `send` 方法时，作为累加器使用的协程可以获取部分结果，不过能返回值的协程更有用。这个特性在 PEP 380 中定义，于 Python 3.3 引入。我们知道，现在生成器中的 `return the_result` 语句会抛出 `StopIteration(the_result)` 异常，这样调用方可以从异常的 `value` 属性中获取 `the_result`。这样获取协程的结果还是很麻烦，不过 PEP 380 引入的 `yield from` 句法能自动处理。

探讨 `yield from` 结构时，我们首先从使用简单的迭代器的示例入手，然后又举了一个例子，重点说明 `yield from` 结构的三个主要组件：委派生成器（在定义体中使用 `yield from`），`yield from` 激活的子生成器，以及通过委派生成器中 `yield from` 表达式架设起来的通道把值发给子生成器，从而驱动整个过程的客户代码。最后，那一节参照 PEP 380 中使用的英语和类似 Python 的伪代码分析了 `yield from` 结构的正

式定义。

本章最后举了一个离散事件仿真示例，说明如何使用生成器代替线程和回调，实现并发。那个出租车仿真系统虽然简单，但是首次一窥了事件驱动型框架（如 Tornado 和 `asyncio`）的运作方式：在单个线程中使用一个主循环驱动协程执行并发活动。使用协程做面向事件编程时，协程会不断把控制权让步给主循环，激活并向前运行其他协程，从而执行各个并发活动。这是一种协作式多任务：协程显式自主地把控制权让步给中央调度程序。而多线程实现的是抢占式多任务。调度程序可以在任何时刻暂停线程（即使在执行一个语句的过程中），把控制权让给其他线程。

最后要说明一点，本章对协程的定义是宽泛的、不正式的，即：通过客户调用 `.send(...)` 方法发送数据或使用 `yield from` 结构驱动的生成器函数。写作本书时，“PEP 342— Coroutines via Enhanced Generators”（https://www.python.org/dev/peps/pep-0342/）和现有的大多数 Python 书籍都使用这个宽泛的定义。第 18 章介绍的 `asyncio` 库建构在协程之上，不过采用的协程定义更为严格：在 `asyncio` 库中，协程（通常）使用 `@asyncio.coroutine` 装饰器装饰，而且始终使用 `yield from` 结构驱动，而不通过直接在协程上调用 `.send(...)` 方法驱动。当然，在 `asyncio` 库的底层，协程使用 `next(...)` 函数和 `.send(...)` 方法驱动，不过在用户代码中只使用 `yield from` 结构驱动协程运行。

16.11 延伸阅读

David Beazley 是 Python 生成器和协程的终极权威。他与 Brian Jones 合著的《Python Cookbook（第 3 版）中文版》一书中有很多使用协程编写的诀窍。Beazley 在 PyCon 期间开设的课程兼有深度和广度，因此享有盛名。首先是 PyCon US 2008 期间的"Generator Tricks for Systems Programmers"课程（http://www.dabeaz.com/generators/），在 PyCon US 2009 期间又开设了声名远播的"A Curious Course on Coroutines and Concurrency"课程（http://www.dabeaz.com/coroutines/，三个部分的全部视频链接很难找到：第一部分，http://pyvideo.org/video/213；第二部分，http://pyvideo.org/video/215；第三部分，http://pyvideo.org/video/214）。他最新的课程在蒙特利尔 PyCon 2014 期间开设，题为"Generators: The Final Frontier"（http://www.dabeaz.com/finalgenerator/）。在这个课程中，他举了更多并发的例子，因此与本书第 18 章的话题联系更大。他根本不担心学员的大脑会爆炸，因此在"The Final Frontier"课程的最后一部分用协程代替了经典的访问者模式，用于计算算术表达式。

使用协程能以多种新方式组织代码，不过与递归和多态（动态调度）一样，要花点时间才能习惯。James Powell 写了一篇文章，题为"Greedy algorithm with coroutines"（http://seriously.dontusethiscode.com/2013/05/01/greedy-coroutine.html）。他在这篇文章中使用协程重写了经典的算法。你可能还想浏览 ActiveState Code 诀窍数据库（https://code.activestate.com/recipes/）中标记为协程的流行诀窍（https://code.activestate.com/recipes/tags/coroutine/）。

Paul Sokolovsky 为 Damien George 开发的超级精简的 MicroPython（http://micropython.org/，针对微控制器）解释器实现了 `yield from` 结构。在研究这个特性的过程中，他制作了非常详细的示意图（https://dl.dropboxusercontent.com/u/44884329/yield-from.pdf），解说 `yield from` 结构的工作原理，并在 python-tulip 邮件列表中分享。Sokolovsky 很友好，允许我把那个 PDF 文件复制到本书的网站中，那个文件的固定链接是 http://flupy.org/resources/yield-from.pdf。

写作本书时，只有 `asyncio` 库本身和使用这个库的代码大量使用 `yield from`。我花了很多时间，想找到不依赖 `asyncio` 库的 `yield from` 示例。Greg Ewing（PEP 380 的作者，为 CPython 实现了 `yield from`）发表了一些 `yield from` 的使用示例（http://www.cosc.canterbury.ac.nz/greg.ewing/python/yield-from/yield_from.html）：`BinaryTree` 类、一个简单的 XML 解析器和一个任务调度程序。

Brett Slatkin 写的《Effective Python：编写高质量 Python 代码的 59 个有效方法》一书中的第 40 条短小精辟，题为“考虑用协程来并发地运行多个函数”（网上有免费的英文版样章，http://www.effectivepython.com/2015/03/10/consider-coroutines-to-run-many-functions-concurrently/）。这一节中使用 `yield from` 驱动生成器的示例是我见过最棒的：那个示例实现了 John Conway 发明的“生命游戏”（https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life），使用协程管理游戏运行过程中各个细胞的状态。该书的随书代码在一个 GitHub 仓库中（https://github.com/bslatkin/effectivepython）。我重构了那个“生命游戏”示例——把 Slatkin 书中的函数和类与测试代码分开（原来的代码：https://github.com/bslatkin/effectivepython/blob/master/example_code/ite
我还编写了 doctest 形式的测试，因此不用运行脚本就能看到各个协程和类的输出。重构后的示例发布在 GitHub Gist 网站上（https://gist.github.com/ramalho/da5590bc38c973408839）。

还有几个有趣的示例没用 `asyncio` 库，只用了 `yield from`：Peter Otten 在 Python Tutor 邮件列表中发布的消息，“Comparing two CSV files using Python”（https://mail.python.org/pipermail/tutor/2015-February/104200.html）；Ian Ward 以 iPython Notebook 形式发布的“Iterables, Iterators, and Generators”教程（http://nbviewer.ipython.org/github/wardi/iterables-iterators-generators/blob/master/Iterables,%20Iterators,%20Generators.ipynb），实现的是剪刀石头布游戏。

Guido van Rossum 在 python-tulip Google Group 中发表了一篇内容很长的消息，题为“The difference between `yield` and `yield-from`”（https://groups.google.com/forum/#!msg/python-tulip/bmphRrryuFk/aB45sEJUomYJ），值得一读。2009 年 3 月 21 日，Nick Coghlan 在 Python-Dev 邮件列表中发布了带有大量注释的 `yield`

from 扩充实现（https://mail.python.org/pipermail/python-dev/2009-March/087382.html）。在那篇消息中，他写道：

> 不管人们是否觉得使用 yield from 结构的代码难以理解，也不管人们能否领会协作式多线程相关的概念，yield from 结构底层的精巧处理能实现真正的嵌套生成器。

Yury Selivanov 撰写的“PEP 492—Coroutines with async and await syntax”（https://www.python.org/dev/peps/pep-0492/）提议为 Python 增加两个关键字：async 和 await。async 与其他现有的关键字结合使用，用于定义新的语言结构。例如，async def 用于定义协程，async for 用于使用异步迭代器（实现 __aiter__ 和 __anext__ 方法，这是协程版的 __iter__ 和 __next__ 方法）迭代可迭代的异步对象。为了避免与即将引入的 async 关键字冲突，asyncio.async() 函数将在 Python 3.4.4 中重命名为 asyncio.ensure_future()。await 关键字的作用与 yield from 结构类似，不过只能在以 async def 定义的协程（禁止使用 yield 和 yield from）中使用。PEP 492 使用新句法把发展成类似协程对象的生成器与全新的原生协程对象明确地区分开了。得益于 async 和 await 关键字，以及几个特殊的新方法，Python 语言将对原生的协程对象提供更好的支持。协程已经做好准备，会成为 Python 未来特别重要的特性，因此 Python 语言应该更好地集成协程。

使用离散事件仿真系统做试验是熟悉协作式多任务的好方法。维基百科中的“Discrete event simulation”一文（https://en.wikipedia.org/wiki/Discrete_event_simulation）是不错的入门资料。[18] Ashish Gupta 写的短篇教程“Writing a Discrete Event Simulation: Ten Easy Lessons”（http://www.cs.northwestern.edu/~agupta/_projects/networking/Que 说明了如何自己动手（不使用特别的库）编写离散事件仿真系统。那篇教程中的代码使用 Java 编写，因此是基于类的，而且没使用协程，不过可以轻松地移植到 Python。除了代码之外，那篇简短的教程还介绍了离散事件仿真的术语和组件。把 Gupta 教程中的示例转换成 Python 类，然后再转换成利用协程的类，是个很好的练习。

[18] 如今，即使终身教授也同意，维基百科几乎是学习任何计算机科学知识的入门首选。对其他知识而言虽然不是如此，但是在计算机科学这方面，维基百科特别棒。

如果想使用现成的 Python 协程库，可以使用 SimPy。这个库的在线文档（https://simpy.readthedocs.org/en/latest/）中说道：

> SimPy 是使用标准的 Python 开发的基于进程的离散事件仿真框架，事件调度程序基于 Python 的生成器实现，因此还可用于异步网络或实现多智能体系统（即可模拟，也可真正通信）。

协程不是特别新的 Python 特性，但是得到异步编程框架支持（Tornado 最先支持）之前，只在较窄的应用领域内使用。Python 3.3 引入的 `yield from` 结构和 Python 3.4 添加的 `asyncio` 包可能会提升协程（和 Python 3.4 本身）的使用量。但写作本书时，Python 3.4 发布还不到一年，因此观看 David Beazley 的课程，阅读涉及这个话题的经典实例时，不会有太多内容深入探讨 Python 协程编程。不过，这只是暂时的。

杂谈

raise from lambda

对编程语言来说，关键字的作用是建立控制流程和表达式计算的基本规则。

语言的关键字像是棋盘游戏中的棋子。对国际象棋来说，关键字是♚、♛、♜、♝、♞和♙；对围棋来说，关键字是●。

国际象棋的棋手实现计划时，有六种类型的棋子可用；而围棋的棋手看起来只有一种类型的棋子可用。可是，在围棋的玩法中，相邻的棋子能构成更大更稳定的棋子，形状各异，不受束缚。围棋棋子的某些排列是不可摧毁的。围棋的表现力比国际象棋强。围棋的开局走法有 361 种，大约有 `1e+170` 个合规的位置；而国际象棋的开局走法有 20 种，有 `1e+50` 个位置。

如果为国际象棋添加一个新棋子，将带来颠覆性的改变；为编程语言添加一个新的关键字也是如此。因此，语言的设计者谨慎考虑引入新关键字是合理的。

表**16-1**：不同编程语言中的关键字数量

关键字数量	语言	备注
5	Smalltalk-80	以句法极简而著称
25	Go	编程语言，而不是围棋*
32	C	指 ANSI C。C99 有37 个关键字，C11 有 44 个
33	Python	Python 2.7 有 31 个关键字，Python 1.5 有 28 个
41	Ruby	关键字可以作为标识符使用（例如，`class` 也是一个方法的名称）
49	Java	与 C 语言一样，基本类型的名称（`char`、`float` 等）是保留字
60	JavaScript	包含 Java 1.0 的所有关键字，很多都没用（http://mzl.la/1JIr8fM）
65	PHP	PHP 5.3 之后引入了七个关键字，如 `goto`、`trait` 和 `yield`
85	C++	据 cppreference.com 网站给出的信息（http://en.cppreference.com/w/cpp/keyword），C++11 在现有的 75 个关键字的基础上添加了 10 个
555	COBOL	这不是我捏造的。参见 IBM ILE COBOL 手册（http://publib.boulder.ibm.com/iseries/v5r2/ic2924/books/x091317316.htm）
∞	Scheme	任何人都能定义新关键字

* 围棋的英文是 Go，因此作者备注这里说的是 Go 语言。——译者注

Python 3 添加了 `nonlocal` 关键字，把 `None`、`True` 和 `False` 提升为关键字，废弃了 `print` 和 `exec`。在语言的发展过程中，弃用关键字十分罕见。表 16-1 列出了几门语言，按照关键字的数量排序。

Scheme 继承了 Lisp 的宏，允许任何人创建特殊的形式，为语言添加新的控制结构和计算规则。用户定义的这种标识符叫作“句法关键字”。Scheme R5RS 标准声称，“这门语言没有保留的标识符”（标准的第 45 页，http://www.schemers.org/Documents/Standards/R5RS/r5rs.pdf），但是 MIT/GNU Scheme（http://www.gnu.org/software/mitscheme/documentation/mit-scheme-ref/Special-Form-Syntax.html#Special-Form-Syntax）这种特殊的实现预定义了 34 个句法关键字，例如 `if`、`lambda` 和 `define-syntax`（用于创建新关键字的关键字）。18

Python 像国际象棋，而 Scheme 像围棋。

现在，回到 Python 句法。我觉得 Guido 对关键字的态度过于保守了。关键字的数量应该少，添加新关键字可能会破坏大量代码，但是在循环中使用 `else` 揭示了一个递归问题：在更适合使用新关键字的地方重用现有的关键字。在 `for`、`while` 和 `try` 的上下文中，应该使用 `then` 关键字，而不该妄用 `else`。

在这个问题上，最严重的一点是重用 `def`。现在，这个关键字用于定义函数、生成器和协程，而这些对象之间的差异很大，不应该使用相同的句法声明。[19]

引入 `yield from` 句法尤其让人失望。再次声明，我觉得真的应该为 Python 使用者提供新的关键字。更糟的是，这开启了新的趋势：把现有的关键字串起来，创建新的句法，而不添加描述性的合理关键字。恐怕有一天我们要苦苦思索 `raise` from `lambda` 是什么意思。

突发新闻

完成本书的技术审校之后，Yury Selivanov 提交的“PEP 492 — Coroutines with async and await

syntax”（https://www.python.org/dev/peps/pep-0492/）好像要被接受了，将在 Python 3.5 中实现。[20]Guido van Rossum 和 Victor Stinner 都支持这个 PEP，前者是 Python 语言的创造者，后者是 `asyncio` 库的主要维护者，而 `asyncio` 库将是新句法的主要使用案例。回应 Selivanov 在 Python-ideas 邮件列表中发布的消息（https://mail.python.org/pipermail/python-ideas/2015-April/033007.html）时，Guido 甚至暗示，为了实现这个 PEP[21]，可能会延迟发布 Python 3.5（https://mail.python.org/pipermail/pythonideas/2015-April/033050.html）。

当然，这会平息前一节所述的大部分抱怨。

[19]“The Value Of Syntax?”一文（http://lambda-the-ultimate.org/node/4295）对可扩展的句法和编程语言的可用性做了有趣的探讨。Lambda the Ultimate 讨论组（http://lambda-the-ultimate.org/）是编程语言极客的度假胜地。

[20]JavaScript、Python 和其他语言都有这样的问题。推荐阅读 Bob Nystrom 写的“What Color Is Your Function?”一文（http://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/）。

[21]Python 3.5 已经接受了 PEP 492，增加了两个关键字：async 和 await。——编者注

第 17 章　使用期物处理并发

> 抨击线程的往往是系统程序员，他们考虑的使用场景对一般的应用程序员来说，也许一生都不会遇到……应用程序员遇到的使用场景，99% 的情况下只需知道如何派生一堆独立的线程，然后用队列收集结果。[1]
>
> ——Michele Simionato
> 深度思考 Python 的人

> [1]摘自 Michele Simionato 发表的文章“Threads, processes and concurrency in Python: some thoughts”（http://www.artima.com/weblogs/viewpost.jsp?thread=299551），副标题为“Removing the hype around the multicore (non) revolution and some (hopefully) sensible comment about threads and other forms of concurrency”。

本章主要讨论 Python 3.2 引入的 `concurrent.futures` 模块，从 PyPI 中安装 `futures` 包（https://pypi.python.org/pypi/futures/）之后，也能在 Python 2.5 及以上版本中使用这个库。这个库封装了前面的引文中 Michele Simionato 所述的模式，特别易于使用。

这一章还会介绍“期物”（future）[2] 的概念。期物指一种对象，表示异步执行的操作。这个概念的作用很大，是 `concurrent.futures` 模块和 `asyncio` 包（第 18 章讨论）的基础。

> [2]“期物”是我自创的词，其中的“物”是指“物件”（object，也就是对象）。起初读者可能不明其意，可与期货、期权和期房对比理解。——译者注

下面举个示例，作为引子。

17.1　示例：网络下载的三种风格

为了高效处理网络 I/O，需要使用并发，因为网络有很高的延迟，所以为了不浪费 CPU 周期去等待，最好在收到网络响应之前做些其他的事。

为了通过代码说明这一点，我写了三个示例程序，从网上下载 20 个国家的国旗图像。第一个示例程序 flags.py 是依序下载的：下载完一个图像，并将其保存在硬盘中之后，才请求下一个图像。另外两个脚本是并发下载的：几乎同时请求所有图像，每下载完一个文件就保存一个文件。flags_threadpool.py 脚本使用 `concurrent.futures` 模块，而 flags_asyncio.py 脚本使用 `asyncio` 包。

示例 17-1 是运行这三个脚本得到的结果，每个脚本都运行三次。我还在 YouTube 上发布了一个 73 秒的视频（https://www.youtube.com/watch?v=A9e9Cy1UkME），让你观看这些脚本的运行情况，你会看到一个 OS X Finder 窗口，显示运行过程中保存的国旗图像文件。这些脚本从 flupy.org 下载图像，而这个网站架设在 CDN 之后，因此第一次运行时可能要等很久才能看到结果。示例 17-1 中显示的结果是运行几次之后收集的，因此 CDN 中已经有了缓存。

示例 **17-1**　运行 flags.py、flags_threadpool.py 和 flags_asyncio.py 脚本得到的结果

```
$ python3 flags.py
BD BR CD CN DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN ❶
20 flags downloaded in 7.26s ❷
$ python3 flags.py
BD BR CD CN DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN
20 flags downloaded in 7.20s
$ python3 flags.py
BD BR CD CN DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN
20 flags downloaded in 7.09s
$ python3 flags_threadpool.py
DE BD CN JP ID EG NG BR RU CD IR MX US PH FR PK VN IN ET TR
20 flags downloaded in 1.37s ❸
$ python3 flags_threadpool.py
EG BR FR IN BD JP DE RU PK PH CD MX ID US NG TR CN VN ET IR
```

```
20 flags downloaded in 1.60s
$ python3 flags_threadpool.py
BD DE EG CN ID RU IN VN ET MX FR CD NG US JP TR PK BR IR PH
20 flags downloaded in 1.22s
$ python3 flags_asyncio.py ❹
BD BR IN ID TR DE CN US IR PK PH FR RU NG VN ET MX EG JP CD
20 flags downloaded in 1.36s
$ python3 flags_asyncio.py
RU CN BR IN FR BD TR EG VN IR PH CD ET ID NG DE JP PK MX US
20 flags downloaded in 1.27s
$ python3 flags_asyncio.py
RU IN ID DE BR VN PK MX US IR ET EG NG BD FR CN JP PH CD TR ❺
20 flags downloaded in 1.42s
```

❶ 每次运行脚本后，首先显示下载过程中下载完毕的国家代码，最后显示一个消息，说明耗时。

❷ flags.py 脚本下载 20 个图像平均用时 7.18 秒。

❸ flags_threadpool.py 脚本平均用时 1.40 秒。

❹ flags_asyncio.py 脚本平均用时 1.35 秒。

❺ 注意国家代码的顺序：对并发下载的脚本来说，每次下载的顺序都不同。

两个并发下载的脚本之间性能差异不大，不过都比依序下载的脚本快 5 倍多。这只是一个特别小的任务，如果把下载的文件数量增加到几百个，并发下载的脚本能比依序下载的脚本快 20 倍或更多。

在公网中测试 HTTP 并发客户端可能不小心变成拒绝服务（Denial-of-Service，DoS）攻击，或者有这么做的嫌疑。我们可以像示例 17-1 那样做，因为那三个脚本被硬编码，限制只发起 20 个请求。如果想大规模测试 HTTP 服务器，应该自己架设测试服务器。在本书的 GitHub 仓库中（https://github.com/fluentpython/example-code），17-futures/countries/README.rst 文件（https://github.com/fluentpython/example-code/blob/master/17-futures/countries/README.rst）说明了如何在本地架设 Nginx 服务

器。

下面我们来分析示例 17-1 测试的两个脚本——flags.py 和 flags_threadpool.py，看看它们的实现方式。第三个脚本 flags_asyncio.py 留到第 18 章再分析。将这三个脚本一起演示是为了表明一个观点：在 I/O 密集型应用中，如果代码写得正确，那么不管使用哪种并发策略（使用线程或 `asyncio` 包），吞吐量都比依序执行的代码高很多。

下面分析代码。

17.1.1 依序下载的脚本

示例 17-2 不太有吸引力，不过实现并发下载的脚本时会重用其中的大部分代码和设置，因此值得分析一下。

为了清楚起见，示例 17-2 没有处理异常。稍后会处理异常，这里我们想集中说明代码的基本结构，以便和并发下载的脚本进行对比。

示例 **17-2** flags.py：依序下载的脚本；另外两个脚本会重用其中几个函数

```
import os
import time
import sys

import requests  ❶

POP20_CC = ('CN IN US ID BR PK NG BD RU JP '
            'MX PH VN ET EG DE IR TR CD FR').split()  ❷

BASE_URL = 'http://flupy.org/data/flags'  ❸

DEST_DIR = 'downloads/'  ❹


def save_flag(img, filename):  ❺
    path = os.path.join(DEST_DIR, filename)
```

```
    with open(path, 'wb') as fp:
        fp.write(img)


def get_flag(cc):  ❻
    url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
    resp = requests.get(url)
    return resp.content


def show(text):  ❼
    print(text, end=' ')
    sys.stdout.flush()


def download_many(cc_list):  ❽
    for cc in sorted(cc_list):  ❾
        image = get_flag(cc)
        show(cc)
        save_flag(image, cc.lower() + '.gif')

    return len(cc_list)


def main(download_many):  ❿
    t0 = time.time()
    count = download_many(POP20_CC)
    elapsed = time.time() - t0
    msg = '\n{} flags downloaded in {:.2f}s'
    print(msg.format(count, elapsed))


if __name__ == '__main__':
    main(download_many)  ⓫
```

❶ 导入 `requests` 库。这个库不在标准库中，因此依照惯例，在导入标准库中的模块（`os`、`time` 和 `sys`）之后导入，而且使用一个空行分隔开。[3]

[3]可以使用 `pip install requests` 命令安装 `requests` 库。——编者注

❷ 列出人口最多的 20 个国家的 ISO 3166 国家代码，按照人口数量降序排列。

❸ 获取国旗图像的网站。[4]

> [4]国旗图像出自 CIA 世界概况（http://1.usa.gov/1JIsmHJ），由美国政府发布，属公共领域。我把这些图像复制到了自己的网站，以此避免向 CIA.gov 发起 DoS 攻击的嫌疑。

❹ 保存图像的本地目录。

❺ 把 `img`（字节序列）保存到 `DEST_DIR` 目录中，命名为 `filename`。

❻ 指定国家代码，构建 URL，然后下载图像，返回响应中的二进制内容。

❼ 显示一个字符串，然后刷新 `sys.stdout`，这样能在一行消息中看到进度。在 Python 中得这么做，因为正常情况下，遇到换行才会刷新 `stdout` 缓冲。

❽ `download_many` 是与并发实现比较的关键函数。

❾ 按字母表顺序迭代国家代码列表，明确表明输出的顺序与输入一致。返回下载的国旗数量。

❿ `main` 函数记录并报告运行 `download_many` 函数之后的耗时。

⓫ `main` 函数必须调用执行下载的函数；我们把 `download_many` 函数当作参数传给 `main` 函数，这样 `main` 函数可以用作库函数，在后面的示例中接收 `download_many` 函数的其他实现。

> Kenneth Reitz 开发的 `requests` 库可通过 PyPI 安装（https://pypi.python.org/pypi/requests），比 Python 3 标准库中的 `urllib.request` 模块更易于使用。其实，`requests` 库提供的 API 更符合 Python 的习惯用法，而且与 Python 2.6 及以上版本兼容。因为 Python 2 中删除了 `urllib2`，Python 3 又使用了其他名称，所以不管使用哪一版 Python，使用 `requests` 库都更方便。

flags.py 脚本中没有什么新知识，只是与其他脚本对比的基准，而且我把它作为一个库使用，避免实现其他脚本时重复编写代码。下面分析使用 `concurrent.futures` 模块重新实现的版本。

17.1.2 使用**concurrent.futures**模块下载

`concurrent.futures` 模块的主要特色是 `ThreadPoolExecutor` 和 `ProcessPoolExecutor` 类，这两个类实现的接口能分别在不同的线程或进程中执行可调用的对象。这两个类在内部维护着一个工作线程或进程池，以及要执行的任务队列。不过，这个接口抽象的层级很高，像下载国旗这种简单的案例，无需关心任何实现细节。

示例 17-3 展示如何使用 `ThreadPoolExecutor.map` 方法，以最简单的方式实现并发下载。

示例 **17-3** flags_threadpool.py：使用 `futures.ThreadPoolExecutor` 类实现多线程下载的脚本

```
from concurrent import futures

from flags import save_flag, get_flag, show, main  ❶

MAX_WORKERS = 20  ❷


def download_one(cc):  ❸
    image = get_flag(cc)
    show(cc)
    save_flag(image, cc.lower() + '.gif')
    return cc


def download_many(cc_list):
    workers = min(MAX_WORKERS, len(cc_list))  ❹
    with futures.ThreadPoolExecutor(workers) as executor:  ❺
        res = executor.map(download_one, sorted(cc_list))  ❻

    return len(list(res))  ❼


if __name__ == '__main__':
    main(download_many)  ❽
```

❶ 重用 `flags` 模块（见示例 17-2）中的几个函数。

❷ 设定 ThreadPoolExecutor 类最多使用几个线程。

❸ 下载一个图像的函数；这是在各个线程中执行的函数。

❹ 设定工作的线程数量：使用允许的最大值（MAX_WORKERS）与要处理的数量之间较小的那个值，以免创建多余的线程。

❺ 使用工作的线程数实例化 ThreadPoolExecutor 类；executor.__exit__ 方法会调用 executor.shutdown(wait=True) 方法，它会在所有线程都执行完毕前阻塞线程。

❻ map 方法的作用与内置的 map 函数类似，不过 download_one 函数会在多个线程中并发调用；map 方法返回一个生成器，因此可以迭代，获取各个函数返回的值。

❼ 返回获取的结果数量；如果有线程抛出异常，异常会在这里抛出，这与隐式调用 next() 函数从迭代器中获取相应的返回值一样。

❽ 调用 flags 模块中的 main 函数，传入 download_many 函数的增强版。

注意，示例 17-3 中的 download_one 函数其实是示例 17-2 中 download_many 函数的 for 循环体。编写并发代码时经常这样重构：把依序执行的 for 循环体改成函数，以便并发调用。

我们用的库叫 concurrency.futures，可是在示例 17-3 中没有见到期物，因此你可能想知道期物在哪里。下一节会解答这个问题。

17.1.3　期物在哪里

期物是 concurrent.futures 模块和 asyncio 包的重要组件，可是，作为这两个库的用户，我们有时却见不到期物。示例 17-3 在背后用到了期物，但是我编写的代码没有直接使用。这一节概述期物，还会举一个例子，展示用法。

从 Python 3.4 起，标准库中有两个名为 Future 的

类：concurrent.futures.Future 和 asyncio.Future。这两个类的作用相同：两个 Future 类的实例都表示可能已经完成或者尚未完成的延迟计算。这与 Twisted 引擎中的 Deferred 类、Tornado 框架中的 Future 类，以及多个 JavaScript 库中的 Promise 对象类似。

期物封装待完成的操作，可以放入队列，完成的状态可以查询，得到结果（或抛出异常）后可以获取结果（或异常）。

我们要记住一件事：通常情况下自己不应该创建期物，而只能由并发框架（concurrent.futures 或 asyncio）实例化。原因很简单：期物表示终将发生的事情，而确定某件事会发生的唯一方式是执行的时间已经排定。因此，只有排定把某件事交给 concurrent.futures.Executor 子类处理时，才会创建 concurrent.futures.Future 实例。例如，Executor.submit() 方法的参数是一个可调用的对象，调用这个方法后会为传入的可调用对象排期，并返回一个期物。

客户端代码不应该改变期物的状态，并发框架在期物表示的延迟计算结束后会改变期物的状态，而我们无法控制计算何时结束。

这两种期物都有 .done() 方法，这个方法不阻塞，返回值是布尔值，指明期物链接的可调用对象是否已经执行。客户端代码通常不会询问期物是否运行结束，而是会等待通知。因此，两个 Future 类都有 .add_done_callback() 方法：这个方法只有一个参数，类型是可调用的对象，期物运行结束后会调用指定的可调用对象。

此外，还有 .result() 方法。在期物运行结束后调用的话，这个方法在两个 Future 类中的作用相同：返回可调用对象的结果，或者重新抛出执行可调用的对象时抛出的异常。可是，如果期物没有运行结束，result 方法在两个 Future 类中的行为相差很大。对 concurrency.futures.Future 实例来说，调用 f.result() 方法会阻塞调用方所在的线程，直到有结果可返回。此时，result 方法可以接收可选的 timeout 参数，如果在指定的时间内期物没有运行完毕，会抛出 TimeoutError 异常。读到 18.1.1 节你会发现，asyncio.Future.result 方法不支持设定超时时间，在那个库中获取期物的结果最好使用 yield from 结构。不过，对 concurrency.futures.Future 实例不能这么做。

这两个库中有几个函数会返回期物，其他函数则使用期物，以用户易于理解的方式实现自身。使用 17-3 中的 Executor.map 方法属于后者：返回值是一个迭代器，迭代器的 __next__ 方法调用各个期物的 result 方法，因此我们得到的是各个期物的结果，而非期物本身。

为了从实用的角度理解期物，我们可以使用 concurrent.futures.as_completed 函数（https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.a 重写示例 17-3。这个函数的参数是一个期物列表，返回值是一个迭代器，在期物运行结束后产出期物。

为了使用 futures.as_completed 函数，只需修改 download_many 函数，把较抽象的 executor.map 调用换成两个 for 循环：一个用于创建并排定期物，另一个用于获取期物的结果。同时，我们会添加几个 print 调用，显示运行结束前后的期物。修改后的 download_many 函数如示例 17-4，代码行数由 5 变成 17，不过现在我们能一窥神秘的期物了。其他函数不变，与示例 17-3 中的一样。

示例 **17-4**　flags_threadpool_ac.py：把download_many 函数中的 executor.map 方法换成 executor.submit 方法和 futures.as_completed 函数

```
def download_many(cc_list):
    cc_list = cc_list[:5]  ❶
    with futures.ThreadPoolExecutor(max_workers=3) as executor:  ❷
        to_do = []
        for cc in sorted(cc_list):  ❸
            future = executor.submit(download_one, cc)  ❹
            to_do.append(future)  ❺
            msg = 'Scheduled for {}: {}'
            print(msg.format(cc, future))  ❻

        results = []
        for future in futures.as_completed(to_do):  ❼
            res = future.result()  ❽
            msg = '{} result: {!r}'
            print(msg.format(future, res))  ❾
            results.append(res)

    return len(results)
```

❶ 这次演示只使用人口最多的 5 个国家。

❷ 把 `max_workers` 硬编码为 `3`，以便在输出中观察待完成的期物。

❸ 按照字母表顺序迭代国家代码，明确表明输出的顺序与输入一致。

❹ `executor.submit` 方法排定可调用对象的执行时间，然后返回一个期物，表示这个待执行的操作。

❺ 存储各个期物，后面传给 `as_completed` 函数。

❻ 显示一个消息，包含国家代码和对应的期物。

❼ `as_completed` 函数在期物运行结束后产出期物。

❽ 获取该期物的结果。

❾ 显示期物及其结果。

注意，在这个示例中调用 `future.result()` 方法绝不会阻塞，因为 `future` 由 `as_completed` 函数产出。运行示例 17-4 得到的输出如示例 17-5 所示。

示例 **17-5**　flags_threadpool_ac.py 脚本的输出

```
$ python3 flags_threadpool_ac.py
Scheduled for BR: <Future at 0x100791518 state=running> ❶
Scheduled for CN: <Future at 0x100791710 state=running>
Scheduled for ID: <Future at 0x100791a90 state=running>
Scheduled for IN: <Future at 0x101807080 state=pending> ❷
Scheduled for US: <Future at 0x101807128 state=pending>
CN <Future at 0x100791710 state=finished returned str> result: 'CN' ❸
BR ID <Future at 0x100791518 state=finished returned str> result: 'BR' ❹
<Future at 0x100791a90 state=finished returned str> result: 'ID'
IN <Future at 0x101807080 state=finished returned str> result: 'IN'
US <Future at 0x101807128 state=finished returned str> result: 'US'

5 flags downloaded in 0.70s
```

❶ 排定的期物按字母表排序；期物的 `repr()` 方法会显示期物的状态：

前三个期物的状态是 `running`，因为有三个工作的线程。

❷ 后两个期物的状态是 `pending`，等待有线程可用。

❸ 这一行里的第一个 `CN` 是运行在一个工作线程中的 `download_one` 函数输出的，随后的内容是 `download_many` 函数输出的。

❹ 这里有两个线程输出国家代码，然后主线程中的 `download_many` 函数输出第一个线程的结果。

> 多次运行 flags_threadpool_ac.py 脚本，看到的结果有所不同。如果把 `max_workers` 参数的值增大到 `5`，结果的顺序变化更多。把 `max_workers` 参数的值设为 `1`，代码依序运行，结果的顺序始终与调用 `submit` 方法的顺序一致。

我们分析了两个版本的使用 `concurrent.futures` 库实现的下载脚本：使用 `ThreadPoolExecutor.map` 方法的示例 17-3 和使用 `futures.as_completed` 函数的示例 17-4。如果你对 flags_asyncio.py 脚本的代码好奇，可以看一眼第 18 章中的示例 18-5。

严格来说，我们目前测试的并发脚本都不能并行下载。使用 `concurrent.futures` 库实现的那两个示例受 GIL（Global Interpreter Lock，全局解释器锁）的限制，而 flags_asyncio.py 脚本在单个线程中运行。

读到这里，你可能会对前面做的非正规基准测试有下述疑问。

- 既然 Python 线程受 GIL 的限制，任何时候都只允许运行一个线程，那么 flags_threadpool.py 脚本的下载速度怎么会比 flags.py 脚本快 5 倍？

- flags_asyncio.py 脚本和 flags.py 脚本都在单个线程中运行，前者怎么会比后者快 5 倍？

第二个问题在 18.3 节解答。

GIL 几乎对 I/O 密集型处理无害，原因参见下一节。

17.2 阻塞型I/O和GIL

CPython 解释器本身就不是线程安全的，因此有全局解释器锁（GIL），一次只允许使用一个线程执行 Python 字节码。因此，一个 Python 进程通常不能同时使用多个 CPU 核心。[5]

> [5]这是 CPython 解释器的局限，与 Python 语言本身无关。Jython 和 IronPython 没有这种限制。不过，目前最快的 Python 解释器 PyPy 也有 GIL。

编写 Python 代码时无法控制 GIL；不过，执行耗时的任务时，可以使用一个内置的函数或一个使用 C 语言编写的扩展释放 GIL。其实，有个使用 C 语言编写的 Python 库能管理 GIL，自行启动操作系统线程，利用全部可用的 CPU 核心。这样做会极大地增加库代码的复杂度，因此大多数库的作者都不这么做。

然而，标准库中所有执行阻塞型 I/O 操作的函数，在等待操作系统返回结果时都会释放 GIL。这意味着在 Python 语言这个层次上可以使用多线程，而 I/O 密集型 Python 程序能从中受益：一个 Python 线程等待网络响应时，阻塞型 I/O 函数会释放 GIL，再运行一个线程。

因此 David Beazley 才说："Python 线程毫无作用。"[6]

> [6]出自"Generators: The Final Frontier"（http://www.dabeaz.com/finalgenerator/），第 106 张幻灯片。

Python 标准库中的所有阻塞型 I/O 函数都会释放 GIL，允许其他线程运行。`time.sleep()` 函数也会释放 GIL。因此，尽管有 GIL，Python 线程还是能在 I/O 密集型应用中发挥作用。

下面简单说明如何在 CPU 密集型作业中使用 `concurrent.futures` 模块轻松绕开 GIL。

17.3　使用concurrent.futures模块启动进程

concurrent.futures 模块的文档（https://docs.python.org/3/library/concurrent.futures.html）副标题是“Launching parallel tasks”（执行并行任务）。这个模块实现的是真正的并行计算，因为它使用 ProcessPoolExecutor 类把工作分配给多个 Python 进程处理。因此，如果需要做 CPU 密集型处理，使用这个模块能绕开 GIL，利用所有可用的 CPU 核心。

ProcessPoolExecutor 和 ThreadPoolExecutor 类都实现了通用的 Executor 接口，因此使用 concurrent.futures 模块能特别轻松地把基于线程的方案转成基于进程的方案。

下载国旗的示例或其他 I/O 密集型作业使用 ProcessPoolExecutor 类得不到任何好处。这一点易于验证，只需把示例 17-3 中下面这几行：

```
def download_many(cc_list):
    workers = min(MAX_WORKERS, len(cc_list))
    with futures.ThreadPoolExecutor(workers) as executor:
```

改成：

```
def download_many(cc_list):
    with futures.ProcessPoolExecutor() as executor:
```

对简单的用途来说，这两个实现 Executor 接口的类唯一值得注意的区别是，ThreadPoolExecutor.__init__ 方法需要 max_workers 参数，指定线程池中线程的数量。在 ProcessPoolExecutor 类中，那个参数是可选的，而且大多数情况下不使用——默认值是 os.cpu_count() 函数返回的 CPU 数量。这样处理说得通，因为对 CPU 密集型的处理来说，不可能要求使用超过 CPU 数量的职程。而对 I/O 密集型处理来说，可以在一个 ThreadPoolExecutor 实例中使用 10

个、100 个或 1000 个线程；最佳线程数取决于做的是什么事，以及可用内存有多少，因此要仔细测试才能找到最佳的线程数。

经过几次测试，我发现使用 `ProcessPoolExecutor` 实例下载 20 面国旗的时间增加到了 1.8 秒，而原来使用 `ThreadPoolExecutor` 的版本是 1.4 秒。主要原因可能是，我的电脑用的是四核 CPU，因此限制只能有 4 个并发下载，而使用线程池的版本有 20 个工作的线程。

`ProcessPoolExecutor` 的价值体现在 CPU 密集型作业上。我用两个 CPU 密集型脚本做了一些性能测试。

arcfour_futures.py

　　这个脚本（代码清单参见示例 A-7）纯粹使用 Python 实现 RC4 算法。我加密并解密了 12 个字节数组，大小从 149KB 到 384KB 不等。

sha_futures.py

　　这个脚本（代码清单参见示例 A-9）使用标准库中的 `hashlib` 模块（使用 OpenSSL 库实现）实现 SHA-256 算法。我计算了 12 个 1MB 字节数组的 SHA-256 散列值。

这两个脚本除了显示汇总结果之外，没有使用 I/O。构建和处理数据的过程都在内存中完成，因此 I/O 对执行时间没有影响。

我运行了 64 次 RC4 示例，48 次 SHA 示例，平均时间如表 17-1 所示。统计的时间中包含派生工作进程的时间。

表**17-1**：在配有**Intel Core i7 2.7 GHz**四核**CPU**的设备中，使用**Python 3.4**运行**RC4**和**SHA**示例，分别使用**1~4**个职程得到的时间和提速倍数

职程数	运行**RC4**示例的时间	**RC4**示例的提速倍数	运行**SHA**示例的时间	**SHA**示例的提速倍数
1	11.48s	1.00×	22.66s	1.00×
2	8.65s	1.33×	14.90s	1.52×

3	6.04s	1.90×	11.91s	1.90×
4	5.58s	2.06×	10.89s	2.08×

可以看出，对加密算法来说，使用 `ProcessPoolExecutor` 类派生 4 个工作的进程后（如果有 4 个 CPU 核心的话），性能可以提高两倍。

对那个纯粹使用 Python 实现的 RC4 示例来说，如果使用 PyPy 和 4 个职程，与使用 CPython 和 4 个职程相比，速度能提高 3.8 倍。以表 17-1 中使用 CPython 和一个职程的运行时间为基准，速度提升了 7.8 倍。

如果使用 Python 处理 CPU 密集型工作，应该试试 PyPy（http://pypy.org）。使用 PyPy 运行 arcfour_futures.py 脚本，速度快了 3.8~5.1 倍；具体的倍数由职程的数量决定。我测试时使用的是 PyPy 2.4.0，这一版与 Python 3.2.5 兼容，因此标准库中有 `concurrent.futures` 模块。

下面通过一个演示程序来研究线程池的行为。这个程序会创建一个包含 3 个职程的线程池，运行 5 个可调用的对象，输出带有时间戳的消息。

17.4 实验Executor.map方法

若想并发运行多个可调用的对象，最简单的方式是使用示例 17-3 中见过的 Executor.map 方法。示例 17-6 中的脚本演示了 Executor.map 方法的某些运作细节。这个脚本的输出在示例 17-7 中。

示例 **17-6** demo_executor_map.py：简单演示 ThreadPoolExecutor 类的 map 方法

```
from time import sleep, strftime
from concurrent import futures


def display(*args):  ❶
    print(strftime('[%H:%M:%S]'), end=' ')
    print(*args)


def loiter(n):  ❷
    msg = '{}loiter({}): doing nothing for {}s...'
    display(msg.format('\t'*n, n, n))
    sleep(n)
    msg = '{}loiter({}): done.'
    display(msg.format('\t'*n, n))
    return n * 10  ❸

def main():
    display('Script starting.')
    executor = futures.ThreadPoolExecutor(max_workers=3)  ❹
    results = executor.map(loiter, range(5))  ❺
    display('results:', results)  ❻
    display('Waiting for individual results:')
    for i, result in enumerate(results):  ❼
        display('result {}: {}'.format(i, result))


main()
```

❶ 这个函数的作用很简单，把传入的参数打印出来，并在前面加上 [HH:MM:SS] 格式的时间戳。

❷ `loiter` 函数什么也没做，只是在开始时显示一个消息，然后休眠 *n* 秒，最后在结束时再显示一个消息；消息使用制表符缩进，缩进的量由 *n* 的值确定。

❸ `loiter` 函数返回 `n * 10`，以便让我们了解收集结果的方式。

❹ 创建 `ThreadPoolExecutor` 实例，有 3 个线程。

❺ 把五个任务提交给 `executor`（因为只有 3 个线程，所以只有 3 个任务会立即开始：`loiter(0)`、`loiter(1)` 和 `loiter(2)`）；这是非阻塞调用。

❻ 立即显示调用 `executor.map` 方法的结果：一个生成器，如示例 17-7 中的输出所示。

❼ `for` 循环中的 `enumerate` 函数会隐式调用 `next(results)`，这个函数又会在（内部）表示第一个任务（`loiter(0)`）的 `_f` 期物上调用 `_f.result()` 方法。`result` 方法会阻塞，直到期物运行结束，因此这个循环每次迭代时都要等待下一个结果做好准备。

我建议你运行示例 17-6，看着结果逐渐显示出来。此外，还可以修改 `ThreadPoolExecutor` 构造方法的 `max_workers` 参数，以及 `executor.map` 方法中 `range` 函数的参数；或者自己挑选几个值，以列表的形式传给 `map` 方法，得到不同的延迟。

示例 17-7 是运行示例 17-6 得到的输出示例。

示例 **17-7**　示例 17-6 中 demo_executor_map.py 脚本的运行示例

```
$ python3 demo_executor_map.py
[15:56:50] Script starting. ❶
[15:56:50] loiter(0): doing nothing for 0s... ❷
[15:56:50] loiter(0): done.
[15:56:50]      loiter(1): doing nothing for 1s... ❸
[15:56:50]              loiter(2): doing nothing for 2s...
[15:56:50] results: <generator object result_iterator at 0x106517168> ❹
[15:56:50]                      loiter(3): doing nothing for 3s... ❺
[15:56:50] Waiting for individual results:
[15:56:50] result 0: 0 ❻
[15:56:51]      loiter(1): done. ❼
```

```
[15:56:51]                              loiter(4): doing nothing for 4s...
[15:56:51] result 1: 10 ❽
[15:56:52]               loiter(2): done. ❾
[15:56:52] result 2: 20
[15:56:53]                      loiter(3): done.
[15:56:53] result 3: 30
[15:56:55]                              loiter(4): done. ❿
[15:56:55] result 4: 40
```

❶ 这次运行从 15:56:50 开始。

❷ 第一个线程执行 `loiter(0)`，因此休眠 0 秒，甚至会在第二个线程开始之前就结束，不过具体情况因人而异。[7]

> [7]具体情况因人而异：对线程来说，你永远不知道某一时刻事件的具体排序；有可能在另一台设备中会看到 `loiter(1)` 在 `loiter(0)` 结束之前开始，这是因为 `sleep` 函数总会释放 GIL。因此，即使休眠 0 秒， Python 也可能会切换到另一个线程。

❸ `loiter(1)` 和 `loiter(2)` 立即开始（因为线程池中有三个职程，可以并发运行三个函数）。

❹ 这一行表明，`executor.map` 方法返回的结果（`results`）是生成器；不管有多少任务，也不管 `max_workers` 的值是多少，目前不会阻塞。

❺ `loiter(0)` 运行结束了，第一个职程可以启动第四个线程，运行 `loiter(3)`。

❻ 此时执行过程可能阻塞，具体情况取决于传给 `loiter` 函数的参数：`results` 生成器的 `__next__` 方法必须等到第一个期物运行结束。此时不会阻塞，因为 `loiter(0)` 在循环开始前结束。注意，这一点之前的所有事件都在同一刻发生——15:56:50。

❼ 一秒钟后，即 15:56:51，`loiter(1)` 运行完毕。这个线程闲置，可以开始运行 `loiter(4)`。

❽ 显示 `loiter(1)` 的结果：10。现在，`for` 循环会阻塞，等待 `loiter(2)` 的结果。

❾ 同上：`loiter(2)` 运行结束，显示结果；`loiter(3)` 也一样。

❿ 2 秒钟后 `loiter(4)` 运行结束，因为 `loiter(4)` 在 15:56:51 时开始，休眠了 4 秒。

`Executor.map` 函数易于使用，不过有个特性可能有用，也可能没用，具体情况取决于需求：这个函数返回结果的顺序与调用开始的顺序一致。如果第一个调用生成结果用时 10 秒，而其他调用只用 1 秒，代码会阻塞 10 秒，获取 `map` 方法返回的生成器产出的第一个结果。在此之后，获取后续结果时不会阻塞，因为后续的调用已经结束。如果必须等到获取所有结果后再处理，这种行为没问题；不过，通常更可取的方式是，不管提交的顺序，只要有结果就获取。为此，要把 `Executor.submit` 方法和 `futures.as_completed` 函数结合起来使用，像示例 17-4 中那样。17.5.2 节会继续讨论这种方式。

> `executor.submit` 和 `futures.as_completed` 这个组合比 `executor.map` 更灵活，因为 `submit` 方法能处理不同的可调用对象和参数，而 `executor.map` 只能处理参数不同的同一个可调用对象。此外，传给 `futures.as_completed` 函数的期物集合可以来自多个 `Executor` 实例，例如一些由 `ThreadPoolExecutor` 实例创建，另一些由 `ProcessPoolExecutor` 实例创建。

下一节根据新的需求继续实现下载国旗的示例，这一次不使用 `executor.map` 方法，而是迭代 `futures.as_completed` 函数返回的结果。

17.5　显示下载进度并处理错误

前面说过，17.1 节中的几个脚本没有处理错误，这样做是为了便于阅读和比较三种方案（依序、多线程和异步）的结构。

为了处理各种错误，我创建了 `flags2` 系列示例。

flags2_common.py

　　这个模块中包含所有 `flags2` 示例通用的函数和设置，例如 `main` 函数，负责解析命令行参数、计时和报告结果。这个脚本中的代码其实是提供支持的，与本章的话题没有直接关系，因此我把源码放在附录 A 里的示例 A-10 中。

flags2_sequential.py

　　能正确处理错误，以及显示进度条的 HTTP 依序下载客户端。flags2_threadpool.py 脚本会用到这个模块里的 `download_one` 函数。

flags2_threadpool.py

　　基于 `futures.ThreadPoolExecutor` 类实现的 HTTP 并发客户端，演示如何处理错误，以及集成进度条。

flags2_asyncio.py

　　与前一个脚本的作用相同，不过使用 `asyncio` 和 `aiohttp` 实现。这个脚本在第 18 章的 18.4 节中分析。



测试并发客户端时要小心

在公开的 HTTP 服务器上测试 HTTP 并发客户端时要小心，因为每秒可能会发起很多请求，这相当于是拒绝服务（DoS）攻击。我们不想攻击任何人，只是在学习如何开发高性能的客户端。访问公开的服务器时一定要管好自己的客户端。做高并发试验时，应该在本

地架设 HTTP 服务器供测试。本书代码仓库中的 17-futures/countries/ 目录里有个 README.rst 文件（https://github.com/fluentpython/example-code/blob/master/17-futures/countries/README.rst），那里有架设说明。

`flags2` 系列示例最明显的特色是，有使用 TQDM 包（https://github.com/noamraph/tqdm）实现的文本动画进度条。我在 YouTube 上发布了一个 108 秒的视频（https://www.youtube.com/watch?v=M8Z65tAl5l4），展示了这个进度条，还对比了三个 `flags` 脚本的下载速度。在那个视频中，我先运行依序下载的脚本，不过 32 秒后中断了，因为那个脚本要用 5 分多钟访问 676 个 URL，下载 194 面国旗；然后，我分别运行多线程和 `asyncio` 版三次，每次都在 6 秒之内（即快了 60 多倍）完成任务。图 17-1 中有两个截图，分别是 flags2_threadpool.py 脚本运行中和运行结束后。

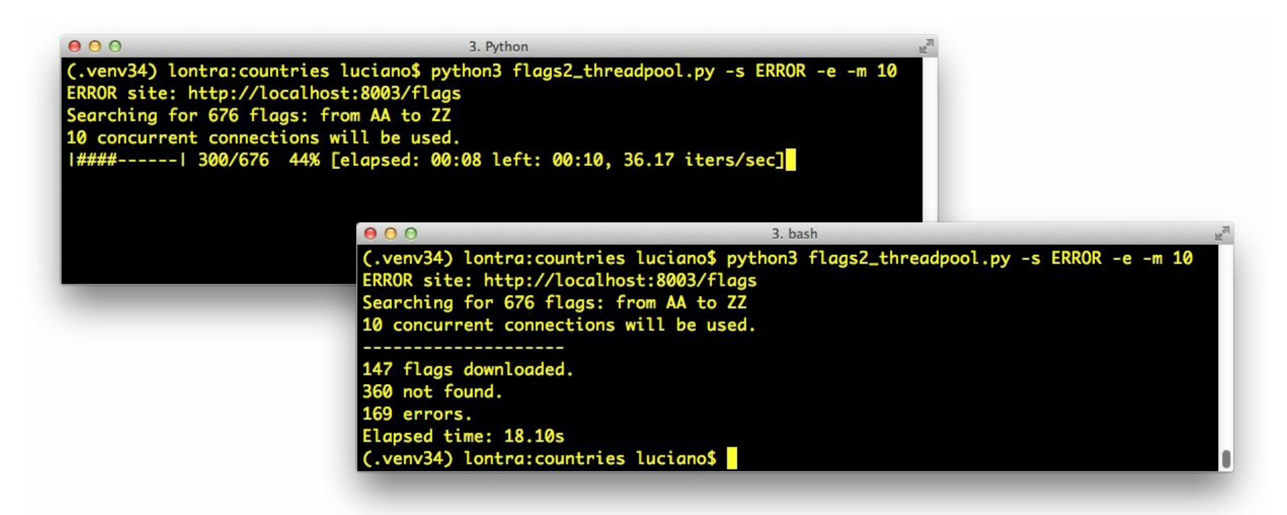


图 17-1：（左上）**flags2_threadpool.py** 运行中，显示着 **tqdm** 包生成的进度条；（右下）同一个终端窗口，脚本运行完毕后

TQDM 包特别易于使用，项目的 README.md 文件（https://github.com/noamraph/tqdm/blob/master/README.md）中有个 GIF 动画，演示了最简单的用法。安装 `tqdm` 包之后，[8] 在 Python 控制台中输入下述代码，会在注释那里看到进度条动画：

[8]可以使用 `pip install tqdm` 命令安装 `tqdm` 包。——编者注

```
>>> import time
>>> from tqdm import tqdm
>>> for i in tqdm(range(1000)):
...     time.sleep(.01)
...
>>> # -> 进度条会出现在这里 <-
```

除了这个灵巧的效果之外，`tqdm` 函数的实现方式也很有趣：能处理任何可迭代的对象，生成一个迭代器；使用这个迭代器时，显示进度条和完成全部迭代预计的剩余时间。为了计算预计剩余时间，`tqdm` 函数要获取一个能使用 `len` 函数确定大小的可迭代对象，或者在第二个参数中指定预期的元素数量。借助在 `flags2` 系列示例中集成 TQDM，我们可以深入了解这几个脚本的运作方式，因为我们必须使用 `futures.as_completed` 函数（https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.a 和 `asyncio.as_completed` 函数（https://docs.python.org/3/library/asyncio-task.html#asyncio.as_completed），这样 `tqdm` 函数才能在每个期物运行结束后更新进度。

`flags2` 系列示例的另一个特色是，提供了命令行接口。三个脚本接受的选项相同，运行任意一个脚本时指定 `-h` 选项就能看到所有选项。示例 17-8 显示的是帮助文本。

示例 **17-8**　`flags2` 系列脚本的帮助界面

```
$ python3 flags2_threadpool.py -h
usage: flags2_threadpool.py [-h] [-a] [-e] [-l N] [-m CONCURRENT] [-s LABEL]
                            [-v]
                            [CC [CC ...]]

Download flags for country codes. Default: top 20 countries by population.

positional arguments:
  CC                    country code or 1st letter (eg. B for BA...BZ)

optional arguments:
  -h, --help            show this help message and exit
  -a, --all             get all available flags (AD to ZW)
  -e, --every           get flags for every possible code (AA...ZZ)
  -l N, --limit N       limit to N first codes
```

```
-m CONCURRENT, --max_req CONCURRENT
                      maximum concurrent requests (default=30)
-s LABEL, --server LABEL
                      Server to hit; one of DELAY, ERROR, LOCAL, REMOTE
                      (default=LOCAL)
-v, --verbose         output detailed progress info
```

所有选项都是可选的。下面说明最重要的选项。

不能忽略的选项是 `-s/--server`：用于选择测试时使用的 HTTP 服务器和基 URL。这个选项的值可以设为下述 4 个字符串（不区分大小写），用于确定脚本从哪里下载国旗。

`LOCAL`

　　使用 `http://localhost:8001/flags`；这是默认值。你应该配置一个本地 HTTP 服务器，响应 8001 端口的请求。我测试时使用 Nginx。本章示例代码中的 README.rst 文件（https://github.com/fluentpython/example-code/blob/master/17-futures/countries/README.rst）说明了如何安装和配置 Nginx。

`REMOTE`

　　使用 `http://flupy.org/data/flags`；这是我搭建的公开网站，托管在一个共享服务器中。请不要使用太多并发请求访问这个网站。`flupy.org` 域名由 Cloudflare CDN（http://www.cloudflare.com/）的一个免费账户管理，因此第一次下载时会发现很慢，不过一旦 CDN 有了缓存，速度就会变快。[9]

> [9]测试这些脚本时，我向那个廉价的虚拟主机发起了一些并发请求，但是得到的响应是“HTTP 503 errors—Service Temporarily Unavailable”。后来我配置了 Cloudflare，现在没有这个错误了。

`DELAY`

　　使用 `http://localhost:8002/flags`；这是一个代理，会延迟 HTTP 响应，监听的端口是 8002。我在本地的 Nginx 服务器前加上了 Mozilla Vaurien，以此引入延迟。前面提到的那个 README.rst 文件（https://github.com/fluentpython/example-code/blob/master/17-

futures/countries/README.rst）中有运行 Vaurien 代理的说明。

`ERROR`

使用 `http://localhost:8003/flags`；这是一个代理，监听 8003 端口，引入了 HTTP 错误，并延迟响应。这个服务器使用的 Vaurien 配置与前面不同。

仅当在本地架设 HTTP 服务器，并且监听 8001 端口时，才能使用 `LOCAL` 选项。`DELAY` 和 `ERROR` 选项需要代理，分别监听 8002 和 8003 端口。在 GitHub 上本书的代码仓库中有个 17-futures/countries/README.rst 文件（https://github.com/fluentpython/example-code/blob/master/17-futures/countries/README.rst），说明了如何配置 Nginx 和 Mozilla Vaurien，以实现这些选项的要求。

默认情况下，各个 `flags2` 脚本会使用默认的并发连接数（各脚本有所不同）从 `LOCAL` 服务器（http://localhost:8001/flags）中下载人口最多的 20 个国家的国旗。示例 17-9 是全部使用默认值运行 flags2_sequential.py 脚本得到的输出。

示例 **17-9**　全部使用默认值运行 flags2_sequential.py 脚本：`LOCAL` 服务器，人口最多的 20 国国旗，1 个并发连接

```
$ python3 flags2_sequential.py
LOCAL site: http://localhost:8001/flags
Searching for 20 flags: from BD to VN
1 concurrent connection will be used.
--------------------
20 flags downloaded.
Elapsed time: 0.10s
```

我们可以使用多种不同的方式选择下载哪些国家的国旗。示例 17-10 展示如何下载国家代码以字母 A、B 或 C 开头的所有国旗。

示例 **17-10**　运行 flags2_threadpool.py 脚本，从 `DELAY` 服务器中下载国家代码以 A、B 或 C 开头的所有国旗

```
$ python3 flags2_threadpool.py -s DELAY a b c
DELAY site: http://localhost:8002/flags
Searching for 78 flags: from AA to CZ
30 concurrent connections will be used.
--------------------
43 flags downloaded.
35 not found.
Elapsed time: 1.72s
```

不管使用什么方式选择国家代码，下载的国旗数量都可以使用 `-l/--limit` 选项限制。示例 17-11 演示如何发起 100 个请求，结合 `-a` 和 `-l` 选项下载 100 面国旗。

示例 **17-11**　运行 flags2_asyncio.py 脚本，使用 100 个并发请求（`-m 100`）从 ERROR 服务器中下载100 面国旗（`-al 100`）

```
$ python3 flags2_asyncio.py -s ERROR -al 100 -m 100
ERROR site: http://localhost:8003/flags
Searching for 100 flags: from AD to LK
100 concurrent connections will be used.
--------------------
73 flags downloaded.
27 errors.
Elapsed time: 0.64s
```

以上是 `flags2` 系列示例的用户界面。下面分析实现方式。

17.5.1　`flags2`系列示例处理错误的方式

这三个示例在负责下载一个文件的函数（`download_one`）中使用相同的策略处理 HTTP 404 错误（未找到）。其他异常则向上冒泡，交给 `download_many` 函数处理。

我们还是先分析依序下载的代码，因为这些代码更易于理解，而且使用线程池的脚本重用了这里的大部分代码。示例 17-12 列出的是 flags2_sequential.py 和 flags2_threadpool.py 脚本真正用于下载的函数。

示例 **17-12**　flags2_sequential.py：负责下载的基本函数；

flags2_threadpool.py 脚本重用了这两个函数

```
def get_flag(base_url, cc):
    url = '{}/{cc}/{cc}.gif'.format(base_url, cc=cc.lower())
    resp = requests.get(url)
    if resp.status_code != 200:  ❶
        resp.raise_for_status()
    return resp.content


def download_one(cc, base_url, verbose=False):
    try:
        image = get_flag(base_url, cc)
    except requests.exceptions.HTTPError as exc:  ❷
        res = exc.response
        if res.status_code == 404:
           status = HTTPStatus.not_found  ❸
           msg = 'not found'
        else:  ❹
            raise
    else:
        save_flag(image, cc.lower() + '.gif')
        status = HTTPStatus.ok
        msg = 'OK'

    if verbose:  ❺
        print(cc, msg)

    return Result(status, cc)  ❻
```

❶ get_flag 函数没有处理错误，当 HTTP 代码不是 200 时，使用 requests.Response.raise_for_status 方法抛出异常。[10]

> [10]HTTP 代码 200 表示成功完成 HTTP 请求。——编者注

❷ download_one 函数捕获 requests.exceptions.HTTPError 异常，特别处理 HTTP 404 错误……

❸ ……方法是，把局部变量 status 设为 HTTPStatus.not_found；HTTPStatus 是从 flags2_common 模块（见示例 A-10）中导入的 Enum 对象。

❹ 重新抛出其他 HTTPError 异常；这些异常会向上冒泡，传给调用方。

❺ 如果在命令行中设定了 -v/--verbose 选项，显示国家代码和状态消息；这就是详细模式中看到的进度信息。

❻ download_one 函数的返回值是一个 namedtuple——Result，其中有个 status 字段，其值是 HTTPStatus.not_found 或 HTTPStatus.ok。

示例 17-13 列出的是 download_many 函数的依序下载版。代码虽然简单，不过值得分析一下，以便后面与并发版对比。我们要关注的是报告进度、处理错误和统计下载数量的方式。

示例 **17-13**　flags2_sequential.py：实现依序下载的 download_many 函数

```
def download_many(cc_list, base_url, verbose, max_req):
    counter = collections.Counter()  ❶
    cc_iter = sorted(cc_list)  ❷
    if not verbose:
        cc_iter = tqdm.tqdm(cc_iter)  ❸
    for cc in cc_iter:  ❹
        try:
            res = download_one(cc, base_url, verbose)  ❺
        except requests.exceptions.HTTPError as exc:  ❻
            error_msg = 'HTTP error {res.status_code} - {res.reason}'
            error_msg = error_msg.format(res=exc.response)
        except requests.exceptions.ConnectionError as exc:  ❼
            error_msg = 'Connection error'
        else:  ❽
            error_msg = ''
            status = res.status

        if error_msg:
            status = HTTPStatus.error  ❾
        counter[status] += 1  ❿
        if verbose and error_msg:  ⓫
            print('*** Error for {}: {}'.format(cc, error_msg))

    return counter  ⓬
```

❶ 这个 Counter 实例用于统计不同的下载状态：HTTPStatus.ok、HTTPStatus.not_found 或 HTTPStatus.error。

❷ 按字母顺序传入的国家代码列表，保存在 cc_iter 变量中。

❸ 如果不是详细模式，把 cc_iter 传给 tqdm 函数，返回一个迭代器，产出 cc_iter 中的元素，还会显示进度条动画。

❹ 这个 for 循环迭代 cc_iter……

❺ ……不断调用 download_one 函数，执行下载。

❻ 处理 get_flag 函数抛出的与 HTTP 有关的且 download_one 函数没有处理的异常。

❼ 处理其他与网络有关的异常。其他异常会中止这个脚本，因为调用 download_many 函数的 flags2_common.main 函数中没有 try/except 块。

❽ 如果没有异常从 download_one 函数中逃出，从 download_one 函数返回的 namedtuple（HTTPStatus）中获取 status。

❾ 如果有错误，把局部变量 status 设为相应的状态。

❿ 以 HTTPStatus（一个 Enum）中的值为键，增加计数器。

⓫ 如果是详细模式，而且有错误，显示带有当前国家代码的错误消息。

⓬ 返回 counter，以便 main 函数能在最终的报告中显示数量。

下面分析重构后的线程池示例——flags2_threadpool.py。

17.5.2　使用**futures.as_completed**函数

为了集成 TQDM 进度条，并处理各次请求中的错误，flags2_threadpool.py 脚本用到我们见过的

`futures.ThreadPoolExecutor` 类和 `futures.as_completed` 函数。示例 17-14 是 flags2_threadpool.py 脚本的完整代码清单。这个脚本只实现了 `download_many` 函数，其他函数都重用 `flags2_common` 和 `flags2_sequential` 模块里的。

示例 **17-14**　flags2_threadpool.py：完整的代码清单

```
import collections
from concurrent import futures

import requests
import tqdm  ❶

from flags2_common import main, HTTPStatus  ❷
from flags2_sequential import download_one  ❸

DEFAULT_CONCUR_REQ = 30  ❹
MAX_CONCUR_REQ = 1000  ❺


def download_many(cc_list, base_url, verbose, concur_req):
    counter = collections.Counter()
    with futures.ThreadPoolExecutor(max_workers=concur_req) as executor:  ❻
        to_do_map = {}  ❼
        for cc in sorted(cc_list):  ❽
            future = executor.submit(download_one,
                            cc, base_url, verbose)  ❾
            to_do_map[future] = cc  ❿
        done_iter = futures.as_completed(to_do_map)  ⓫
        if not verbose:
            done_iter = tqdm.tqdm(done_iter, total=len(cc_list))  ⓬
        for future in done_iter:  ⓭
            try:
                res = future.result()  ⓮
            except requests.exceptions.HTTPError as exc:  ⓯
                error_msg = 'HTTP {res.status_code} - {res.reason}'
                error_msg = error_msg.format(res=exc.response)
            except requests.exceptions.ConnectionError as exc:
                error_msg = 'Connection error'
            else:
                error_msg = ''
                status = res.status

            if error_msg:
                status = HTTPStatus.error
```

```
            counter[status] += 1
            if verbose and error_msg:
                cc = to_do_map[future]  ⓰
                print('*** Error for {}: {}'.format(cc, error_msg))
    return counter


if __name__ == '__main__':
    main(download_many, DEFAULT_CONCUR_REQ, MAX_CONCUR_REQ)
```

❶ 导入显示进度条的库。

❷ 从 flags2_common 模块中导入一个函数和一个 Enum。

❸ 重用 flags2_sequential 模块（见示例 17-12）里的 download_one 函数。

❹ 如果没有在命令行中指定 -m/--max_req 选项，使用这个值作为并发请求数的最大值，也就是线程池的大小；真实的数量可能会比这少，例如下载的国旗数量较少。

❺ 不管要下载多少国旗，也不管 -m/--max_req 命令行选项的值是多少，MAX_CONCUR_REQ 会限制最大的并发请求数；这是一项安全预防措施。

❻ 把 max_workers 设为 concur_req，创建 ThreadPoolExecutor 实例；main 函数会把下面这三个值中最小的那个赋值给 concur_req：MAX_CONCUR_REQ、cc_list 的长度、-m/--max_req 命令行选项的值。这样能避免创建超过所需的线程。

❼ 这个字典把各个 Future 实例（表示一次下载）映射到相应的国家代码上，在处理错误时使用。

❽ 按字母顺序迭代国家代码列表。结果的顺序主要由 HTTP 响应的时间长短决定，不过，如果线程池的大小（由 concur_req 设定）比 len(cc_list) 小得多，可能会发现有按字母顺序批量下载的情况。

❾ 每次调用 executor.submit 方法排定一个可调用对象的执行时间，然后返回一个 Future 实例。第一个参数是可调用的对象，其余的参数

是传给可调用对象的参数。

❿ 把返回的 future 和国家代码存储在字典中。

⓫ futures.as_completed 函数返回一个迭代器，在期物运行结束后产出期物。

⓬ 如果不是详细模式，把 as_completed 函数返回的结果传给 tqdm 函数，显示进度条；因为 done_iter 没有 len 函数，所以我们必须通过 total= 参数告诉 tqdm 函数预期的元素数量，这样 tqdm 才能预计剩余的工作量。

⓭ 迭代运行结束后的期物。

⓮ 在期物上调用 result 方法，要么返回可调用对象的返回值，要么抛出可调用的对象在执行过程中捕获的异常。这个方法可能会阻塞，等待确定结果；不过，在这个示例中不会阻塞，因为 as_completed 函数只返回已经运行结束的期物。

⓯ 处理可能出现的异常；这个函数余下的代码与依序下载版 download_many 函数一样（见示例 17-13），不过下一点除外。

⓰ 为了给错误消息提供上下文，以当前的 future 为键，从 to_do_map 中获取国家代码。在依序下载版中无须这么做，因为那一版迭代的是国家代码，所以知道当前国家的代码；而这里迭代的是期物。

示例 17-14 用到了一个对 futures.as_completed 函数特别有用的惯用法：构建一个字典，把各个期物映射到其他数据（期物运行结束后可能有用）上。这里，在 to_do_map 中，我们把各个期物映射到对应的国家代码上。这样，尽管期物生成的结果顺序已经乱了，依然便于使用结果做后续处理。

Python 线程特别适合 I/O 密集型应用，concurrent.futures 模块大大简化了某些使用场景下 Python 线程的用法。我们对 concurrent.futures 模块基本用法的介绍到此结束。下面讨论不适合使用 ThreadPoolExecutor 或 ProcessPoolExecutor 类时，有哪些替代方案。

17.5.3　线程和多进程的替代方案

Python 自 0.9.8 版（1993 年）就支持线程了，concurrent.futures 只不过是使用线程的最新方式。Python 3 废弃了原来的 thread 模块，换成了高级的 threading 模块（https://docs.python.org/3/library/threading.html）。[11] 如果 futures.ThreadPoolExecutor 类对某个作业来说不够灵活，可能要使用 threading 模块中的组件（如 Thread、Lock、Semaphore 等）自行制定方案，比如说使用 queue 模块（https://docs.python.org/3/library/queue.html）创建线程安全的队列，在线程之间传递数据。futures.ThreadPoolExecutor 类已经封装了这些组件。

> [11]threading 模块自 Python 1.5.1（1998 年）就已存在，不过有些人仍然继续使用旧的 thread 模块。Python 3 把 thread 模块重命名为 _thread，以此强调这是低层实现，不应该在应用代码中使用。

对 CPU 密集型工作来说，要启动多个进程，规避 GIL。创建多个进程最简单的方式是，使用 futures.ProcessPoolExecutor 类。不过和前面一样，如果使用场景较复杂，需要更高级的工具。multiprocessing 模块（https://docs.python.org/3/library/multiprocessing.html）的 API 与 threading 模块相仿，不过作业交给多个进程处理。对简单的程序来说，可以用 multiprocessing 模块代替 threading 模块，少量改动即可。不过，multiprocessing 模块还能解决协作进程遇到的最大挑战：在进程之间传递数据。

17.6　本章小结

本章开头对两个 HTTP 并发客户端和一个依序下载的客户端做了对比，结果是并发版比依序下载的脚本性能高很多。

分析过使用 `concurrent.futures` 实现的第一个示例后，我们深入探讨了期物对象，即 `concurrent.futures.Future` 或 `asyncio.Future` 类的实例，着重说明了二者的共同点（区别在第 18 章详述）。我们说明了如何使用 `Executor.submit(...)` 方法创建期物，以及如何使用 `concurrent.futures.as_completed(...)` 函数迭代运行结束的期物。

接下来，我们分析了为什么尽管有 GIL，Python 线程仍然适合 I/O 密集型应用：标准库中每个使用 C 语言编写的 I/O 函数都会释放 GIL，因此，当某个线程在等待 I/O 时， Python 调度程序会切换到另一个线程。然后，我们讨论了如何借助 `concurrent.futures.ProcessPoolExecutor` 类使用多进程，以此绕开 GIL，使用多个 CPU 核心运行加密算法，并通过四个职程实现一倍多的速度提升。

在随后的一节中，我们深入分析了 `concurrent.futures.ThreadPoolExecutor` 类的运作方式。为了说明问题，我特意举了一个示例，创建几个任务，但是休眠几秒钟，什么也不做，只是显示带有时间戳的状态。

接下来，本章回到下载国旗的示例，增加了进度条和错误处理代码，并且进一步探索了 `future.as_completed` 生成器函数。我们得知一个常见的做法：把期物存储在一个字典中，提交期物时把期物与相关的信息联系起来；这样，`as_completed` 迭代器产出期物后，就可以使用那些信息。

最后，本章简要说明了多线程和多进程并发的低层实现（但却更灵活）——`threading` 和 `multiprocessing` 模块。这两个模块代表在 Python 中使用线程和进程的传统方式。

17.7　延伸阅读

Brian Quinlan 是 `concurrent.futures` 包的贡献者，他在 PyCon Australia 2010 上所做的“The Future Is Soon!”（http://www.pyvideo.org/video/480/pyconau-2010--the-future-is-soon）演讲对这个包做了介绍。Quinlan 演讲时没用幻灯片，而是直接在 Python 控制台中输入代码，以此说明这个库的用途。作为引子，他在演讲中推荐了 XKCD 漫画家和程序员 Randall Munroe 制作的一个视频，Randall 在这个视频中对 Google Maps 发起了 DoS 攻击（非有意为之），绘制一个彩色地图，显示他驾车绕城的路线。这个库的正式介绍文件是“PEP 3148—`futures`—execute computations asynchronously”（https://www.python.org/dev/peps/pep-3148/）。在这个 PEP 中，Quinlan 写道，`concurrent.futures` 库“受 Java 的 `java.util.concurrent` 包影响很大”。

Jan Palach 写的 *Parallel Programming with Python*（Packt 出版社）一书介绍了几个并发编程的工具，包括 `concurrent.futures`、`threading` 和 `multiprocessing` 库。除了标准库之外，这本书还讨论了 Celery（http://celery.readthedocs.org/en/latest/getting-started/introduction.html）。这是一个任务队列，用于把工作分配给多个线程和进程，甚至是不同的设备。在 Django 社区中，为了减轻繁重任务的负担（例如，把生成 PDF 的工作交给其他进程，防止 HTTP 响应延迟生成），Celery 可能是使用最广泛的系统。

Beazley 与 Jones 的著作《Python Cookbook（第 3 版）中文版》有多个使用 `concurrent.futures` 的诀窍，首先是“11.12 理解事件驱动型 I/O”。“12.7 创建线程池”展示了一个简单的 TCP 回显服务器，“12.8 实现简单的并行编程”提供了一个特别实用的示例：借助 `ProcessPoolExecutor` 实例分析一整个目录中使用 `gzip` 压缩的 Apache 日志文件。这本书的第 12 章对线程做了更多介绍，特别值得一提的是“12.10 定义一个 Actor 任务”，这个诀窍演示了参与者模型：通过传递消息协调多个线程的可行方式。

Brett Slatkin 写的《Effective Python：编写高质量 Python 代码的 59 个有效方法》一书中有一章探讨了并发的多个话题，包括：协程；使用

`concurrent.futures` 库处理线程和进程；不使用 `ThreadPoolExecutor` 类，而使用锁和队列做线程编程。

Micha Gorelick 与 Ian Ozsvald 写的 *High Performance Python*（O'Reilly 出版社）和 Doug Hellmann 写的《Python 标准库》都涵盖了线程和进程。

若想了解不使用线程或回调的现代并发方式，推荐阅读 Paul Butcher 写的《七周七并发模型》。[12] 我喜欢这本书的副标题“When Threads Unravel”（线程束手无策之时）。这本书的第 1 章简单介绍了线程和锁，后面的六章探讨了不同语言（不包括 Python、Ruby 和 JavaScript）为并发编程提供的现代化替代方案。

> [12]该书已由人民邮电出版社出版，书号：978-7-115-38606-9。——编者注

如果对 GIL 感兴趣，请先阅读 Python 文档中的“Python Library and Extension FAQ”（“Can't we get rid of the Global Interpreter Lock?”，https://docs.python.org/3/faq/library.html#id18）。Guido van Rossum 写的“It isn't Easy to Remove the GIL”（http://www.artima.com/weblogs/viewpost.jsp?thread=214235）和 Jesse Noller（`multiprocessing` 包的贡献者）写的“Python Threads and the Global Interpreter Lock”（http://jessenoller.com/2009/02/01/python-threads-and-the-global-interpreter-lock/）也值得一读。此外，David Beazley 在“Understanding the Python GIL”（http://www.dabeaz.com/GIL/）[13] 中详细探讨了 GIL 的内部运作。在这次演讲的第 54 张幻灯片中（http://www.dabeaz.com/python/UnderstandingGIL.pdf），Beazley 得出了一些令人担忧的结果，例如，使用 Python 3.2 引入的新 GIL 算法做基准测试时，他发现处理时间增加了 20 倍。不过，Beazley 似乎使用一个空的 `while True: pass` 循环模拟 CPU 密集型工作，而现实中不会这样做。在 Beazley 提交的缺陷报告中，根据 Antoine Pitrou（实现新 GIL 算法的人）的评论（http://bugs.python.org/issue7946#msg223110），这个问题与工作负载没有太大关系。

> [13]感谢 Lucas Brunialti 把这个演讲的链接发给我。

GIL 是实际存在的问题，而且短时间内不可能消失，不过 Jesse Noller 和 Richard Oudkerk 开发了一个库，能让 CPU 密集型应用轻松地绕开这个问题——`multiprocessing` 包。这个包在多个进程中模拟 `threading`

模块的 API，而且支持基础设施的锁、队列、管道、共享内存，等等。这个包由“PEP 371—Addition of the multiprocessing package to the standard library”（https://www.python.org/dev/peps/pep-0371/）引入。这个包的官方文档是个 93KB 的 .rst 文件（大约 63 页），是 Python 标准库文档中最长的一章。多进程是 `concurrent.futures.ProcessPoolExecutor` 类的基础。

对于 CPU 密集型和数据密集型并行处理，现在有个新工具可用——分布式计算引擎 Apache Spark（https://spark.apache.org/）。Spark 在大数据领域发展势头强劲，提供了友好的 Python API，支持把 Python 对象当作数据，如示例页面所示（https://spark.apache.org/examples.html）。

João S. O. Bueno 开发的 `lelo` 库（https://pypi.python.org/pypi/lelo）和 Nat Pryce 开发的 `python-parallelize` 库（https://github.com/npryce/python-parallelize）简洁且十分易于使用，它们的作用是使用多个进程处理并行任务。`lelo` 包定义了一个 `@parallel` 装饰器，可以应用到任何函数上，把函数变成非阻塞：调用被装饰的函数时，函数在一个新进程中执行。 Nat Pryce 开发的 `python-parallelize` 包提供了一个 `parallelize` 生成器，能把 `for` 循环分配给多个 CPU 执行。这两个包在内部都使用了 `multiprocessing` 模块。

杂谈

远离线程

> 并发是计算机科学中最难的概念之一（通常最好别去招惹它）。[14]
>
> ——David Beazley
> Python 教练和科学狂人

上面引自 David Beazley 的话与本章开头引自 Michele Simionato 的话明显矛盾，但我都同意。在大学学过一门并发课程之后（那门课把“并发编程”与管理线程和锁划上等号），我得出一个结论，我不该自己管理线程和锁，而应该管理内存分配和释放。线程和锁最好由懂行的系统程序员管理，他们有这种爱好，也有时间去管理（但愿如此）。

因此我觉得 `concurrent.futures` 包很棒，它把线程、进程和队列视作服务的基础设施，不用自己动手直接处理。当然，这个包针对的是简单的作业，也就是所谓的“高度并行”问题（https://en.wikipedia.org/wiki/Embarrassingly_parallel）。可是，正如本章开头 Simionato 所说的那样，编写应用（而非操作系统或数据库服务器）时，遇到的大部分并发问题都属于这一种。

对于并发程度不高的问题来说，线程和锁也不是解决之道。在操作系统层面，线程永远不会消失；不过，过去七年我觉得让人眼前一亮的编程语言（包括 Go、Elixir 和 Clojure）都对并发做了更好、更高层的抽象，正如《七周七并发模型》一书所述。Erlang（实现 Elixir 的语言）是典型示例，设计这门语言时彻底考虑到了并发。我对这门语言不感兴趣的原因很简单——句法丑陋。我被 Python 的句法宠坏了。

José Valim 是著名的 Ruby on Rails 核心贡献者，他设计的 Elixir 提供了友好而现代的句法。与 Lisp 和 Clojure 一样，Elixir 也实现了句法宏。这是把双刃剑。使用句法宏能实现强大的 DSL，可是衍生语言多起来之后，代码基会出现兼容问题，社区会分裂。大量涌现的宏导致 Lisp 没落，因为各种 Lisp 实现都使用独特难懂的方言。标准化的 Common Lisp 则开始复苏。我希望 José Valim 能引领 Elixir 社区，不要重蹈覆辙。

与 Elixir 类似，Go 也是一门充满新意的现代语言。可是，与 Elixir 相比，某些方面有点保守。Go 不支持宏，句法比 Python 简单。Go 也不支持继承和运算符重载，而且提供的元编程支持没有 Python 多。这些限制被认为是 Go 语言的特点，因为行为和性能更可预料。这对高并发来说是好事，而 Go 的重要使命是取代 C++、Java 和 Python。

虽然 Elixir 和 Go 在高并发领域是直接的竞争者，但是设计原理的不同则吸引了不同的用户群。这两门语言都可能蓬勃发展。可是纵观编程语言的历史，保守的语言更能吸引程序员。我希望自己能精通 Go 和 Elixir。

关于 **GIL**

GIL 简化了 CPython 解释器和 C 语言扩展的实现。得益于 GIL，

Python 有很多 C 语言扩展——这绝对是如今 Python 如此受欢迎的主要原因之一。

多年以来，我一直觉得 GIL 导致 Python 线程几乎没有用武之地，只能开发一些玩具应用。直到发现标准库中每一个阻塞型 I/O 函数都会释放 GIL 之后，我才意识到 Python 线程特别适合在 I/O 密集型系统（鉴于我的工作经验，客户经常付费让我开发这种应用）中使用。

竞争对手对并发的支持

MRI（推荐使用的 Ruby 实现）也有 GIL，因此，Ruby 线程与 Python 线程受到同样的限制。相比之下，JavaScript 解释器则根本不支持用户层级的线程。在 JavaScript 中，只能通过回调式异步编程实现并发。我提到这些是因为，Ruby 和 JavaScript 是最能直接与 Python 竞争的通用动态编程语言。

在深谙并发的这一批新语言中，Go 和 Elixir 或许是最能蚕食 Python 的语言。不过，现在有 `asyncio` 了。既然这么多人相信纯粹使用回调的 Node.js 平台可以做并发编程，那么 `asyncio` 生态系统成熟后，Python 赢回这些人能有多难呢？不过，这是下一章“杂谈”的话题。

[14]摘自 PyCon 2009 教程“A Curious Course on Coroutines and Concurrency”（http://www.dabeaz.com/coroutines/）的第 9 张幻灯片。

第 18 章　使用 asyncio 包处理并发

> 并发是指一次处理多件事。
> 并行是指一次做多件事。
> 二者不同，但是有联系。
> 一个关于结构，一个关于执行。
> 并发用于制定方案，用来解决可能（但未必）并行的问题。[1]
>
> ——Rob Pike
> Go 语言的创造者之一

> [1]摘自“Concurrency Is Not Parallelism (It's Better)”（http://concur.rspace.googlecode.com/hg/talk/concur.html#slide-5）演讲的第 5 张幻灯片。

Imre Simon 教授 [2] 说过，科学界有两个重要过错：使用不同的词表示相同的事物，以及使用同一个词表示不同的事物。如果你研究过并发编程或并行编程，会发现“并发”和“并行”有不同的定义。我将采用上述引文中 Rob Pike 的非正式定义。

> [2]Imre Simon（1943—2009）是巴西的计算机科学先驱，对自动机理论（Automata Theory）有杰出的贡献，开创了热带数学（Tropical Mathematics）这一领域。他还是自由软件和自由文化的拥护者。我有幸曾与他一起学习、工作和相处。

真正的并行需要多个核心。现代的笔记本电脑有 4 个 CPU 核心，但是通常不经意间就有超过 100 个进程同时运行。因此，实际上大多数过程都是并发处理的，而不是并行处理。计算机始终运行着 100 多个进程，确保每个进程都有机会取得进展，不过 CPU 本身同时做的事情不能超过四件。十年前使用的设备也能并发处理 100 个进程，不过都在同一个核心里。鉴于此，Rob Pike 才把那次演讲取名为“Concurrency Is Not Parallelism (It's Better)”[“并发不是并行（并发更好）”]。

本章介绍 `asyncio` 包，这个包使用事件循环驱动的协程实现并发。这是 Python 中最大也是最具雄心壮志的库之一。Guido van Rossum 在

Python 仓库之外开发 `asyncio` 包，把这个项目的代号命名为“Tulip”（郁金香）。因此，在网上搜索这方面的资料时，会经常看到这种花的名称。例如，这个项目的主要讨论组仍叫 python-tulip（https://groups.google.com/forum/#!forum/python-tulip）。

Python 3.4 把 Tulip 添加到标准库中时，把它重命名为 `asyncio`。这个包也兼容 Python 3.3，在 PyPI 中可以通过新的官方名称找到（https://pypi.python.org/pypi/asyncio）。`asyncio` 大量使用 `yield from` 表达式，因此与 Python 旧版不兼容。

> Trollius 项目（也以花名命名，http://trollius.readthedocs.org/）移植了 `asyncio`，把 `yield from` 替换成 `yield` 和精巧的回调（`From` 和 `Return`），以便支持 Python 2.6 及以上版本。`yield from ...` 表达式变成了 `yield From(...)`；如果协程需要返回结果，那么要把 `return result` 替换成 `raise Return(result)`。Trollius 由 Victor Stinner 主导，他也是 `asyncio` 包的核心开发者。Victor 人很好，在本书付梓之前同意审核本章。

本章讨论以下话题：

- 对比一个简单的多线程程序和对应的 `asyncio` 版，说明多线程和异步任务之间的关系
- `asyncio.Future` 类与 `concurrent.futures.Future` 类之间的区别
- 第 17 章中下载国旗那些示例的异步版
- 摒弃线程或进程，如何使用异步编程管理网络应用中的高并发
- 在异步编程中，与回调相比，协程显著提升性能的方式
- 如何把阻塞的操作交给线程池处理，从而避免阻塞事件循环
- 使用 `asyncio` 编写服务器，重新审视 Web 应用对高并发的处理方式

- 为什么 `asyncio` 已经准备好对 Python 生态系统产生重大影响

首先，本章通过简单的示例来对比 `threading` 模块和 `asyncio` 包。

18.1　线程与协程对比

有一次讨论线程和 GIL 时，Michele Simionato 发布了一个简单但有趣的示例（https://mail.python.org/pipermail/python-list/2009-February/525280.html）：在长时间计算的过程中，使用 `multiprocessing` 包在控制台中显示一个由 ASCII 字符 `"|/-\"` 构成的动画旋转指针。

我改写了 Simionato 的示例，一个借由 `threading` 模块使用线程实现，一个借由 `asyncio` 包使用协程实现。我这么做是为了让你对比两种实现，理解如何不使用线程来实现并发行为。

示例 18-1 和示例 18-2 的输出是动态的，因此你一定要运行这两个脚本，看看结果如何。如果你在坐地铁（或者在某个没有 Wi-Fi 连接的地方），可以看图 18-1，想象单词“thinking”之前的 \ 线是旋转的。

```
2. Python
(.venv34) lontra:17-concurrency luciano$ python3 spinner_thread.py
spinner object: <Thread(Thread-1, initial)>
Answer: 42
(.venv34) lontra:17-concurrency luciano$ python3 spinner_asyncio.py
spinner object: <Task pending coro=<spin() running at spinner_asyncio.py:12>>
\ thinking!
```

图 **18-1**：**spinner_thread.py** 和 **spinner_asyncio.py** 两个脚本的输出类似：旋转指针对象的字符串表示形式和文本**“Answer: 42”**。在这个截图中，**spinner_asyncio.py** 脚本仍在运行中，旋转指针显示的是**“\ thinking!”**消息；脚本运行结束后，那一行会替换成**“Answer: 42”**

首先，分析 spinner_thread.py 脚本（见示例 18-1）。

示例 **18-1**　spinner_thread.py：通过线程以动画形式显示文本式旋转指针

```
import threading
import itertools
import time
import sys


class Signal:  ❶
    go = True


def spin(msg, signal):  ❷
    write, flush = sys.stdout.write, sys.stdout.flush
    for char in itertools.cycle('|/-\\'):  ❸
        status = char + ' ' + msg
        write(status)
        flush()
        write('\x08' * len(status))  ❹
        time.sleep(.1)
        if not signal.go:  ❺
            break
    write(' ' * len(status) + '\x08' * len(status))  ❻


def slow_function():  ❼
    # 假装等待I/O一段时间
    time.sleep(3)  ❽
    return 42


def supervisor():  ❾
    signal = Signal()
    spinner = threading.Thread(target=spin,
                               args=('thinking!', signal))
    print('spinner object:', spinner)  ❿
    spinner.start()  ⓫
    result = slow_function()  ⓬
    signal.go = False  ⓭
    spinner.join()  ⓮
    return result


def main():
    result = supervisor()  ⓯
    print('Answer:', result)


if __name__ == '__main__':
```

```
    main()
```

❶ 这个类定义一个简单的可变对象；其中有个 go 属性，用于从外部控制线程。

❷ 这个函数会在单独的线程中运行。signal 参数是前面定义的 Signal 类的实例。

❸ 这其实是个无限循环，因为 itertools.cycle 函数会从指定的序列中反复不断地生成元素。

❹ 这是显示文本式动画的诀窍所在：使用退格符（\x08）把光标移回来。

❺ 如果 go 属性的值不是 True 了，那就退出循环。

❻ 使用空格清除状态消息，把光标移回开头。

❼ 假设这是耗时的计算。

❽ 调用 sleep 函数会阻塞主线程，不过一定要这么做，以便释放 GIL，创建从属线程。

❾ 这个函数设置从属线程，显示线程对象，运行耗时的计算，最后杀死线程。

❿ 显示从属线程对象。输出类似于 <Thread(Thread-1, initial)>。

⓫ 启动从属线程。

⓬ 运行 slow_function 函数，阻塞主线程。同时，从属线程以动画形式显示旋转指针。

⓭ 改变 signal 的状态；这会终止 spin 函数中的那个 for 循环。

⓮ 等待 spinner 线程结束。

⓯ 运行 `supervisor` 函数。

注意，Python 没有提供终止线程的 API，这是有意为之的。若想关闭线程，必须给线程发送消息。这里，我使用的是 `signal.go` 属性：在主线程中把它设为 `False` 后，`spinner` 线程最终会注意到，然后干净地退出。

下面来看如何使用 `@asyncio.coroutine` 装饰器替代线程，实现相同的行为。

> 第 16 章的小结说过，`asyncio` 包使用的“协程”是较严格的定义。适合 `asyncio` API 的协程在定义体中必须使用 `yield from`，而不能使用 `yield`。此外，适合 `asyncio` 的协程要由调用方驱动，并由调用方通过 `yield from` 调用；或者把协程传给 `asyncio` 包中的某个函数，例如 `asyncio.async(...)` 和本章要介绍的其他函数，从而驱动协程。最后，`@asyncio.coroutine` 装饰器应该应用在协程上，如下述示例所示。

我们来分析示例 18-2。

示例 **18-2**　spinner_asyncio.py：通过协程以动画形式显示文本式旋转指针

```
import asyncio
import itertools
import sys


@asyncio.coroutine  ❶
def spin(msg):  ❷
    write, flush = sys.stdout.write, sys.stdout.flush
    for char in itertools.cycle('|/-\\'):
        status = char + ' ' + msg
        write(status)
        flush()
        write('\x08' * len(status))
        try:
            yield from asyncio.sleep(.1)  ❸
        except asyncio.CancelledError:  ❹
            break
```

```
    write(' ' * len(status) + '\x08' * len(status))


@asyncio.coroutine
def slow_function():  ❺
    # 假装等待I/O一段时间
    yield from asyncio.sleep(3)  ❻
    return 42


@asyncio.coroutine
def supervisor():  ❼
    spinner = asyncio.async(spin('thinking!'))  ❽
    print('spinner object:', spinner)  ❾
    result = yield from slow_function()  ❿
    spinner.cancel()  ⓫
    return result


def main():
    loop = asyncio.get_event_loop()  ⓬
    result = loop.run_until_complete(supervisor())  ⓭
    loop.close()
    print('Answer:', result)


if __name__ == '__main__':
    main()
```

❶ 打算交给 `asyncio` 处理的协程要使用 `@asyncio.coroutine` 装饰。这不是强制要求，但是强烈建议这么做。原因在本列表后面。

❷ 这里不需要示例 18-1 中 `spin` 函数中用来关闭线程的 `signal` 参数。

❸ 使用 `yield from asyncio.sleep(.1)` 代替 `time.sleep(.1)`，这样的休眠不会阻塞事件循环。

❹ 如果 `spin` 函数苏醒后抛出 `asyncio.CancelledError` 异常，其原因是发出了取消请求，因此退出循环。

❺ 现在，`slow_function` 函数是协程，在用休眠假装进行 I/O 操作时，使用 `yield from` 继续执行事件循环。

❻ yield from asyncio.sleep(3) 表达式把控制权交给主循环，在休眠结束后恢复这个协程。

❼ 现在，supervisor 函数也是协程，因此可以使用 yield from 驱动 slow_function 函数。

❽ asyncio.async(...) 函数排定 spin 协程的运行时间，使用一个 Task 对象包装 spin 协程，并立即返回。

❾ 显示 Task 对象。输出类似于 <Task pending coro=<spin() running at spinner_ asyncio.py:12>>。

❿ 驱动 slow_function() 函数。结束后，获取返回值。同时，事件循环继续运行，因为 slow_function 函数最后使用 yield from asyncio.sleep(3) 表达式把控制权交回给了主循环。

⓫ Task 对象可以取消；取消后会在协程当前暂停的 yield 处抛出 asyncio.CancelledError 异常。协程可以捕获这个异常，也可以延迟取消，甚至拒绝取消。

⓬ 获取事件循环的引用。

⓭ 驱动 supervisor 协程，让它运行完毕；这个协程的返回值是这次调用的返回值。

> 除非想阻塞主线程，从而冻结事件循环或整个应用，否则不要在 asyncio 协程中使用 time.sleep(...)。如果协程需要在一段时间内什么也不做，应该使用 yield from asyncio.sleep(DELAY)。

使用 @asyncio.coroutine 装饰器不是强制要求，但是强烈建议这么做，因为这样能在一众普通的函数中把协程凸显出来，也有助于调试：如果还没从中产出值，协程就被垃圾回收了（意味着有操作未完成，因此有可能是个缺陷），那就可以发出警告。这个装饰器不会预激协程。

注意，spinner_thread.py 和 spinner_asyncio.py 两个脚本的代码行数差不多。supervisor 函数是这两个示例的核心。下面详细对比二者。示例

18-3 只列出了线程版示例中的 `supervisor` 函数。

示例 **18-3**　spinner_thread.py：线程版 `supervisor` 函数

```
def supervisor():
    signal = Signal()
    spinner = threading.Thread(target=spin,
                               args=('thinking!', signal))
    print('spinner object:', spinner)
    spinner.start()
    result = slow_function()
    signal.go = False
    spinner.join()
    return result
```

为了对比，示例 18-4 列出了 `supervisor` 协程。

示例 **18-4**　spinner_asyncio.py：异步版 `supervisor` 协程

```
@asyncio.coroutine
def supervisor():
    spinner = asyncio.async(spin('thinking!'))
    print('spinner object:', spinner)
    result = yield from slow_function()
    spinner.cancel()
    return result
```

这两种 `supervisor` 实现之间的主要区别概述如下。

- `asyncio.Task` 对象差不多与 `threading.Thread` 对象等效。Victor Stinner（本章的特约技术审校）指出，“`Task` 对象像是实现协作式多任务的库（例如 `gevent`）中的绿色线程（green thread）”。

- `Task` 对象用于驱动协程，`Thread` 对象用于调用可调用的对象。

- `Task` 对象不由自己动手实例化，而是通过把协程传给 `asyncio.async(...)` 函数或 `loop.create_task(...)` 方法获取。

- 获取的 `Task` 对象已经排定了运行时间（例如，由 `asyncio.async` 函数排定）；`Thread` 实例则必须调用 `start` 方法，明确告知让它运行。

- 在线程版 `supervisor` 函数中，`slow_function` 函数是普通的函数，直接由线程调用。在异步版 `supervisor` 函数中，`slow_function` 函数是协程，由 `yield from` 驱动。

- 没有 API 能从外部终止线程，因为线程随时可能被中断，导致系统处于无效状态。如果想终止任务，可以使用 `Task.cancel()` 实例方法，在协程内部抛出 `CancelledError` 异常。协程可以在暂停的 `yield` 处捕获这个异常，处理终止请求。

- `supervisor` 协程必须在 `main` 函数中由 `loop.run_until_complete` 方法执行。

上述比较应该能帮助你理解，与更熟悉的 `threading` 模型相比，`asyncio` 是如何编排并发作业的。

线程与协程之间的比较还有最后一点要说明：如果使用线程做过重要的编程，你就知道写出程序有多么困难，因为调度程序任何时候都能中断线程。必须记住保留锁，去保护程序中的重要部分，防止多步操作在执行的过程中中断，防止数据处于无效状态。

而协程默认会做好全方位保护，以防止中断。我们必须显式产出才能让程序的余下部分运行。对协程来说，无需保留锁，在多个线程之间同步操作，协程自身就会同步，因为在任意时刻只有一个协程运行。想交出控制权时，可以使用 `yield` 或 `yield from` 把控制权交还调度程序。这就是能够安全地取消协程的原因：按照定义，协程只能在暂停的 `yield` 处取消，因此可以处理 `CancelledError` 异常，执行清理操作。

下面说明 `asyncio.Future` 类与第 17 章所用的 `concurrent.futures.Future` 类之间的区别。

18.1.1　`asyncio.Future`：故意不阻塞

`asyncio.Future` 类与 `concurrent.futures.Future` 类的接口基本一

致，不过实现方式不同，不可以互换。“PEP 3156—Asynchronous IO Support Rebooted: the‘asyncio’Module”（https://www.python.org/dev/peps/pep-3156/）对这个不幸状况是这样说的：

> 未来可能会统一 `asyncio.Future` 和 `concurrent.futures.Future` 类实现的期物（例如，为后者添加兼容 `yield from` 的 `__iter__` 方法）。

如 17.1.3 节所述，期物只是调度执行某物的结果。在 `asyncio` 包中，`BaseEventLoop.create_task(...)` 方法接收一个协程，排定它的运行时间，然后返回一个 `asyncio.Task` 实例——也是 `asyncio.Future` 类的实例，因为 `Task` 是 `Future` 的子类，用于包装协程。这与调用 `Executor.submit(...)` 方法创建 `concurrent.futures.Future` 实例是一个道理。

与 `concurrent.futures.Future` 类似，`asyncio.Future` 类也提供了 `.done()`、`.add_done_callback(...)` 和 `.result()` 等方法。前两个方法的用法与 17.1.3 节所述的一样，不过 `.result()` 方法差别很大。

`asyncio.Future` 类的 `.result()` 方法没有参数，因此不能指定超时时间。此外，如果调用 `.result()` 方法时期物还没运行完毕，那么 `.result()` 方法不会阻塞去等待结果，而是抛出 `asyncio.InvalidStateError` 异常。

然而，获取 `asyncio.Future` 对象的结果通常使用 `yield from`，从中产出结果，如示例 18-8 所示。

使用 `yield from` 处理期物，等待期物运行完毕这一步无需我们关心，而且不会阻塞事件循环，因为在 `asyncio` 包中，`yield from` 的作用是把控制权还给事件循环。

注意，使用 `yield from` 处理期物与使用 `add_done_callback` 方法处理协程的作用一样：延迟的操作结束后，事件循环不会触发回调对象，而是设置期物的返回值；而 `yield from` 表达式则在暂停的协程中生成返回值，恢复执行协程。

总之，因为 asyncio.Future 类的目的是与 yield from 一起使用，所以通常不需要使用以下方法。

- 无需调用 my_future.add_done_callback(...)，因为可以直接把想在期物运行结束后执行的操作放在协程中 yield from my_future 表达式的后面。这是协程的一大优势：协程是可以暂停和恢复的函数。

- 无需调用 my_future.result()，因为 yield from 从期物中产出的值就是结果（例如，result = yield from my_future）。

当然，有时也需要使用 .done()、.add_done_callback(...) 和 .result() 方法。但是一般情况下，asyncio.Future 对象由 yield from 驱动，而不是靠调用这些方法驱动。

下面分析 yield from 和 asyncio 包的 API 如何拉近期物、任务和协程的关系。

18.1.2　从期物、任务和协程中产出

在 asyncio 包中，期物和协程关系紧密，因为可以使用 yield from 从 asyncio.Future 对象中产出结果。这意味着，如果 foo 是协程函数（调用后返回协程对象），抑或是返回 Future 或 Task 实例的普通函数，那么可以这样写：res = yield from foo()。这是 asyncio 包的 API 中很多地方可以互换协程与期物的原因之一。

为了执行这些操作，必须排定协程的运行时间，然后使用 asyncio.Task 对象包装协程。对协程来说，获取 Task 对象有两种主要方式。

asyncio.async(coro_or_future, *, loop=None)

　　这个函数统一了协程和期物：第一个参数可以是二者中的任何一个。如果是 Future 或 Task 对象，那就原封不动地返回。如果是协程，那么 async 函数会调用 loop.create_task(...) 方法创建 Task 对象。loop= 关键字参数是可选的，用于传入事件循环；如果没有传入，那么 async 函数会通过调用 asyncio.get_event_loop() 函数获

取循环对象。

`BaseEventLoop.create_task(coro)`

　　这个方法排定协程的执行时间，返回一个 `asyncio.Task` 对象。如果在自定义的 `BaseEventLoop` 子类上调用，返回的对象可能是外部库（如 Tornado）中与 `Task` 类兼容的某个类的实例。

> `BaseEventLoop.create_task(...)` 方法只在 Python 3.4.2 及以上版本中可用。如果是 Python 3.3 或 Python 3.4 的旧版，要使用 `asyncio.async(...)` 函数，或者从 PyPI 中安装较新的 `asyncio` 版本（https://pypi.python.org/pypi/asyncio）。

`asyncio` 包中有多个函数会自动（内部使用的是 `asyncio.async` 函数）把参数指定的协程包装在 `asyncio.Task` 对象中，例如 `BaseEventLoop.run_until_complete(...)` 方法。

如果想在 Python 控制台或者小型测试脚本中试验期物和协程，可以使用下述代码片段：[3]

> [3]摘自 Petr Viktorin 于 2014 年 9 月 11 日在 Python-ideas 邮件列表中发布的消息（https://mail.python.org/pipermail/python-ideas/2014-September/029294.html）。

```
>>> import asyncio
>>> def run_sync(coro_or_future):
...     loop = asyncio.get_event_loop()
...     return loop.run_until_complete(coro_or_future)
...
>>> a = run_sync(some_coroutine())
```

在 `asyncio` 包的文档中，“18.5.3. Tasks and coroutines”一节（https://docs.python.org/3/library/asyncio-task.html）说明了协程、期物和任务之间的关系。其中有个注解说道：

> 这份文档把一些方法说成是协程，即使它们其实是返回 `Future` 对象的普通 Python 函数。这是故意的，为的是给以后修改这些函数的实现留下余地。

掌握这些基础知识后，接下来要分析异步下载国旗的 flags_asyncio.py 脚本。这个脚本的用法在示例 17-1（第 17 章）中与依序下载版和线程池版一同演示过。

18.2　使用asyncio和aiohttp包下载

从 Python 3.4 起，asyncio 包只直接支持 TCP 和 UDP。如果想使用 HTTP 或其他协议，那么要借助第三方包。当下，使用 asyncio 实现 HTTP 客户端和服务器时，使用的似乎都是 aiohttp 包。

示例 18-5 是下载国旗的 flags_asyncio.py 脚本的完整代码清单。运作方式简述如下。

(1) 首先，在 download_many 函数中获取一个事件循环，处理调用 download_one 函数生成的几个协程对象。

(2) asyncio 事件循环依次激活各个协程。

(3) 客户代码中的协程（如 get_flag）使用 yield from 把职责委托给库里的协程（如 aiohttp.request）时，控制权交还事件循环，执行之前排定的协程。

(4) 事件循环通过基于回调的低层 API，在阻塞的操作执行完毕后获得通知。

(5) 获得通知后，主循环把结果发给暂停的协程。

(6) 协程向前执行到下一个 yield from 表达式，例如 get_flag 函数中的 yield from resp.read()。事件循环再次得到控制权，重复第 4~6 步，直到事件循环终止。

这与 16.9.2 节所见的示例类似。在那个示例中，主循环依次启动多个出租车进程；各个出租车进程产出结果后，主循环调度各个出租车的下一个事件（未来发生的事），然后激活队列中的下一个出租车进程。那个出租车仿真简单得多，主循环易于理解。不过，在 asyncio 中，基本的流程是一样的：在一个单线程程序中使用主循环依次激活队列里的协程。各个协程向前执行几步，然后把控制权让给主循环，主循环再激活队列里的下一个协程。

下面详细分析示例 18-5。

示例 **18-5** flags_asyncio.py：使用 `asyncio` 和 `aiohttp` 包实现的异步下载脚本

```
import asyncio

import aiohttp  ❶

from flags import BASE_URL, save_flag, show, main  ❷


@asyncio.coroutine  ❸
def get_flag(cc):
    url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
    resp = yield from aiohttp.request('GET', url)  ❹
    image = yield from resp.read()  ❺
    return image


@asyncio.coroutine
def download_one(cc):  ❻
    image = yield from get_flag(cc)  ❼
    show(cc)
    save_flag(image, cc.lower() + '.gif')
    return cc


def download_many(cc_list):
    loop = asyncio.get_event_loop()  ❽
    to_do = [download_one(cc) for cc in sorted(cc_list)]  ❾
    wait_coro = asyncio.wait(to_do)  ❿
    res, _ = loop.run_until_complete(wait_coro)  ⓫
    loop.close() ⓬

    return len(res)


if __name__ == '__main__':
    main(download_many)
```

❶ 必须安装 `aiohttp` 包，它不在标准库中。[4]

[4]可以使用 `pip install aiohttp` 命令安装 `aiohttp` 包。——编者注

❷ 重用 `flags` 模块（见示例 17-2）中的一些函数。

❸ 协程应该使用 `@asyncio.coroutine` 装饰。

❹ 阻塞的操作通过协程实现，客户代码通过 `yield from` 把职责委托给协程，以便异步运行协程。

❺ 读取响应内容是一项单独的异步操作。

❻ `download_one` 函数也必须是协程，因为用到了 `yield from`。

❼ 与依序下载版 `download_one` 函数唯一的区别是这一行中的 `yield from`；函数定义体中的其他代码与之前完全一样。

❽ 获取事件循环底层实现的引用。

❾ 调用 `download_one` 函数获取各个国旗，然后构建一个生成器对象列表。

❿ 虽然函数的名称是 `wait`，但它不是阻塞型函数。`wait` 是一个协程，等传给它的所有协程运行完毕后结束（这是 `wait` 函数的默认行为；参见这个示例后面的说明）。

⓫ 执行事件循环，直到 `wait_coro` 运行结束；事件循环运行的过程中，这个脚本会在这里阻塞。我们忽略 `run_until_complete` 方法返回的第二个元素。下文说明原因。

⓬关闭事件循环。

> 如果事件循环是上下文管理器就好了，这样我们就可以使用 `with` 块确保循环会被关闭。然而，实际情况是复杂的，客户代码绝不会直接创建事件循环，而是调用 `asyncio.get_event_loop()` 函数，获取事件循环的引用。而且有时我们的代码不“拥有”事件循环，因此关闭事件循环会出错。例如，使用 Quamash（https://pypi.python.org/pypi/Quamash/）这种包实现的外部 GUI 事件循环时，Qt 库负责在退出应用时关闭事件循环。

asyncio.wait(...) 协程的参数是一个由期物或协程构成的可迭代对象；wait 会分别把各个协程包装进一个 Task 对象。最终的结果是，wait 处理的所有对象都通过某种方式变成 Future 类的实例。wait 是协程函数，因此返回的是一个协程或生成器对象；wait_coro 变量中存储的正是这种对象。为了驱动协程，我们把协程传给 loop.run_until_complete(...) 方法。

loop.run_until_complete 方法的参数是一个期物或协程。如果是协程，run_until_complete 方法与 wait 函数一样，把协程包装进一个 Task 对象中。协程、期物和任务都能由 yield from 驱动，这正是 run_until_complete 方法对 wait 函数返回的 wait_coro 对象所做的事。wait_coro 运行结束后返回一个元组，第一个元素是一系列结束的期物，第二个元素是一系列未结束的期物。在示例 18-5 中，第二个元素始终为空，因此我们把它赋值给 _，将其忽略。但是 wait 函数有两个关键字参数，如果设定了可能会返回未结束的期物；这两个参数是 timeout 和 return_when。详情参见 asyncio.wait 函数的文档（https://docs.python.org/3/library/asyncio-task.html#asyncio.wait）。

注意，在示例 18-5 中不能重用 flags.py 脚本（见示例 17-2）中的 get_flag 函数，因为那个函数用到了 requests 库，执行的是阻塞型 I/O 操作。为了使用 asyncio 包，我们必须把每个访问网络的函数改成异步版，使用 yield from 处理网络操作，这样才能把控制权交还给事件循环。在 get_flag 函数中使用 yield from，意味着它必须像协程那样驱动。

因此，不能重用 flags_threadpool.py 脚本（见示例 17-3）中的 download_one 函数。示例 18-5 中的代码使用 yield from 驱动 get_flag 函数，因此 download_one 函数本身也得是协程。每次请求时，download_many 函数会创建一个 download_one 协程对象；这些协程对象先使用 asyncio.wait 协程包装，然后由 loop.run_until_complete 方法驱动。

asyncio 包中有很多新概念要掌握，不过，如果你采用 Guido van Rossum 建议的一个技巧，就能轻松地理解示例 18-5 的总体逻辑：眯着眼，假装没有 yield from 关键字。这样做之后，你会发现示例 18-5 中的代码与纯粹依序下载的代码一样易于阅读。

比如说，以这个协程为例：

```
@asyncio.coroutine
def get_flag(cc):
    url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
    resp = yield from aiohttp.request('GET', url)
    image = yield from resp.read()
    return image
```

我们可以假设它与下述函数的作用相同，只不过协程版从不阻塞：

```
def get_flag(cc):
    url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
    resp = aiohttp.request('GET', url)
    image = resp.read()
    return image
```

`yield from foo` 句法能防止阻塞，是因为当前协程（即包含 `yield from` 代码的委派生成器）暂停后，控制权回到事件循环手中，再去驱动其他协程；`foo` 期物或协程运行完毕后，把结果返回给暂停的协程，将其恢复。

在 16.7 节的末尾，我对 `yield from` 的用法做了两点陈述，摘要如下。

- 使用 `yield from` 链接的多个协程最终必须由不是协程的调用方驱动，调用方显式或隐式（例如，在 `for` 循环中）在最外层委派生成器上调用 `next(...)` 函数或 `.send(...)` 方法。

- 链条中最内层的子生成器必须是简单的生成器（只使用 `yield`）或可迭代的对象。

在 `asyncio` 包的 API 中使用 `yield from` 时，这两点都成立，不过要注意下述细节。

- 我们编写的协程链条始终通过把最外层委派生成器传给 `asyncio` 包 API 中的某个函数（如 `loop.run_until_complete(...)`）驱动。

也就是说，使用 `asyncio` 包时，我们编写的代码不通过调用 `next(...)` 函数或 `.send(...)` 方法驱动协程——这一点由 `asyncio` 包实现的事件循环去做。

- 我们编写的协程链条最终通过 `yield from` 把职责委托给 `asyncio` 包中的某个协程函数或协程方法（例如示例 18-2 中的 `yield from asyncio.sleep(...)`），或者其他库中实现高层协议的协程（例如示例 18-5 中 `get_flag` 协程里的 `resp = yield from aiohttp. request('GET', url)`）。

  也就是说，最内层的子生成器是库中真正执行 I/O 操作的函数，而不是我们自己编写的函数。

概括起来就是：使用 `asyncio` 包时，我们编写的异步代码中包含由 `asyncio` 本身驱动的协程（即委派生成器），而生成器最终把职责委托给 `asyncio` 包或第三方库（如 `aiohttp`）中的协程。这种处理方式相当于架起了管道，让 `asyncio` 事件循环（通过我们编写的协程）驱动执行低层异步 I/O 操作的库函数。

现在可以回答第 17 章提出的那个问题了。

- flags_asyncio.py 脚本和 flags.py 脚本都在单个线程中运行，前者怎么会比后者快 5 倍？

18.3　避免阻塞型调用

Ryan Dahl（Node.js 的发明者）在介绍他的项目背后的哲学时说："我们处理 I/O 的方式彻底错了。"[5] 他把执行硬盘或网络 I/O 操作的函数定义为阻塞型函数，主张不能像对待非阻塞型函数那样对待阻塞型函数。为了说明原因，他展示了表 18-1 中的前两列。

> [5]"Introduction to Node.js"（https://www.youtube.com/watch?v=M-sc73Y-zQA）视频 4:55 处。

表**18-1**：使用现代的电脑从不同的存储介质中读取数据的延迟情况；第三栏按比例换算成具体的时间，便于人类理解

存储介质	**CPU** 周期	按比例换算成"人类时间"
L1 缓存	3	3 秒
L2 缓存	14	14 秒
RAM	250	250 秒
硬盘	41 000 000	1.3 年
网络	240 000 000	7.6 年

为了理解表 18-1，请记住一点：现代的 CPU 拥有 GHz 数量级的时钟频率，每秒钟能运行几十亿个周期。假设 CPU 每秒正好运行十亿个周期，那么 CPU 可以在一秒钟内读取 L1 缓存 333 333 333 次，读取网络 4 次（只有 4 次）。表 18-1 中的第三栏是拿第二栏中的各个值乘以固定的因子得到的。因此，在另一个世界中，如果读取 L1 缓存要用 3 秒，那么读取网络要用 7.6 年！

有两种方法能避免阻塞型调用中止整个应用程序的进程：

- 在单独的线程中运行各个阻塞型操作
- 把每个阻塞型操作转换成非阻塞的异步调用使用

多个线程是可以的，但是各个操作系统线程（Python 使用的是这种线程）消耗的内存达兆字节（具体的量取决于操作系统种类）。如果要处理几千个连接，而每个连接都使用一个线程的话，我们负担不起。

为了降低内存的消耗，通常使用回调来实现异步调用。这是一种低层概念，类似于所有并发机制中最古老、最原始的那种——硬件中断。使用回调时，我们不等待响应，而是注册一个函数，在发生某件事时调用。这样，所有调用都是非阻塞的。因为回调简单，而且消耗低，所以 Ryan Dahl 拥护这种方式。

当然，只有异步应用程序底层的事件循环能依靠基础设置的中断、线程、轮询和后台进程等，确保多个并发请求能取得进展并最终完成，这样才能使用回调。[6] 事件循环获得响应后，会回过头来调用我们指定的回调。不过，如果做法正确，事件循环和应用代码共用的主线程绝不会阻塞。

> [6]其实，虽然 Node.js 不支持使用 JavaScript 编写的用户级线程，但是在背后却借助 `libeio` 库使用 C 语言实现了线程池，以此提供基于回调的文件 API——因为从 2014 年起，大多数操作系统都不提供稳定且便携的异步文件处理 API 了。

把生成器当作协程使用是异步编程的另一种方式。对事件循环来说，调用回调与在暂停的协程上调用 `.send()` 方法效果差不多。各个暂停的协程是要消耗内存，但是比线程消耗的内存数量级小。而且，协程能避免可怕的“回调地狱”；这一点会在 18.5 节讨论。

现在你应该能理解为什么 flags_asyncio.py 脚本的性能比 flags.py 脚本高 5 倍了：flags.py 脚本依序下载，而每次下载都要用几十亿个 CPU 周期等待结果。其实，CPU 同时做了很多事，只是没有运行你的程序。与此相比，在 flags_asyncio.py 脚本中，在 `download_many` 函数中调用 `loop.run_until_complete` 方法时，事件循环驱动各个 `download_one` 协程，运行到第一个 `yield from` 表达式处，那个表达式又驱动各个 `get_flag` 协程，运行到第一个 `yield from` 表达式处，

调用 aiohttp.request(...) 函数。这些调用都不会阻塞，因此在零点几秒内所有请求全部开始。

asyncio 的基础设施获得第一个响应后，事件循环把响应发给等待结果的 get_flag 协程。得到响应后，get_flag 向前执行到下一个 yield from 表达式处，调用 resp.read() 方法，然后把控制权还给主循环。其他响应会陆续返回（因为请求几乎同时发出）。所有 get_ flag 协程都获得结果后，委派生成器 download_one 恢复，保存图像文件。

为了尽量提高性能，save_flag 函数应该执行异步操作，可是 asyncio 包目前没有提供异步文件系统 API（Node 有）。如果这是应用的瓶颈，可以使用 loop.run_in_executor 方法（https://docs.python.org/3/library/asyncio-eventloop.html#asyncio.BaseEventLoop.run_in_executor），在线程池中运行 save_flag 函数。示例 18-9 会说明做法。

因为异步操作是交叉执行的，所以并发下载多张图像所需的总时间比依序下载少得多。我使用 asyncio 包发起了 600 个 HTTP 请求，获得所有结果的时间比依序下载快 70 倍。

现在回到那个 HTTP 客户端示例，看看如何显示动态的进度条，并且恰当地处理错误。

18.4　改进asyncio下载脚本

17.5 节说过，`flags2` 系列示例的命令行接口相同。本节要分析这个系列中的 flags2_asyncio.py 脚本。例如，示例 18-6 展示如何使用 100 个并发请求（`-m 100`）从 ERROR 服务器中下载 100 面国旗（`-al 100`）。

示例 **18-6**　运行 flags2_asyncio.py 脚本

```
$ python3 flags2_asyncio.py -s ERROR -al 100 -m 100
ERROR site: http://localhost:8003/flags
Searching for 100 flags: from AD to LK
100 concurrent connections will be used.
--------------------
73 flags downloaded.
27 errors.
Elapsed time: 0.64s
```



测试并发客户端要谨慎

尽管线程版和 `asyncio` 版 HTTP 客户端的下载总时间相差无几，但是 `asyncio` 版发送请求的速度更快，因此很有可能对服务器发起 DoS 攻击。为了全速测试这些并发客户端，应该在本地搭建 HTTP 服务器，详情参见本书代码仓库（https://github.com/fluentpython/example-code）中 17-futures/countries/ 目录里的 README.rst 文件（https://github.com/fluentpython/example-code/blob/master/17-futures/countries/README.rst）。

下面分析 flags2_asyncio.py 脚本的实现方式。

18.4.1　使用asyncio.as_completed函数

在示例 18-5 中，我把一个协程列表传给 `asyncio.wait` 函数，经由 `loop.run_until_complete` 方法驱动，全部协程运行完毕后，这个函数会返回所有下载结果。可是，为了更新进度条，各个协程运行结束后

就要立即获取结果。在线程池版示例中（见示例 17-14），为了集成进度条，我们使用的是 as_completed 生成器函数；幸好，asyncio 包提供了这个生成器函数的相应版本。

为了使用 asyncio 包实现 flags2 示例，我们要重写几个函数；重写后的函数可以供 concurrent.future 版重用。之所以要重写，是因为在使用 asyncio 包的程序中只有一个主线程，而在这个线程中不能有阻塞型调用，因为事件循环也在这个线程中运行。所以，我要重写 get_flag 函数，使用 yield from 访问网络。现在，由于 get_flag 是协程， download_one 函数必须使用 yield from 驱动它，因此 download_one 自己也要变成协程。之前，在示例 18-5 中，download_one 由 download_many 驱动：download_one 函数由 asyncio. wait 函数调用，然后传给 loop.run_until_complete 方法。现在，为了报告进度并处理错误，我们要更精确地控制，所以我把 download_many 函数中的大多数逻辑移到一个新的协程 downloader_coro 中，只在 download_many 函数中设置事件循环，以及调度 downloader_coro 协程。

示例 18-7 展示的是 flags2_asyncio.py 脚本的前半部分，定义 get_flag 和 download_one 协程。示例 18-8 列出余下的源码，定义 downloader_coro 协程和 download_many 函数。

示例 **18-7**　flags2_asyncio.py：脚本的前半部分；余下的代码在示例 18-8 中

```
import asyncio
import collections

import aiohttp
from aiohttp import web
import tqdm

from flags2_common import main, HTTPStatus, Result, save_flag

# 默认设为较小的值，防止远程网站出错
# 例如503 - Service Temporarily Unavailable
DEFAULT_CONCUR_REQ = 5
MAX_CONCUR_REQ = 1000
```

```
class FetchError(Exception):  ❶
    def __init__(self, country_code):
        self.country_code = country_code


@asyncio.coroutine
def get_flag(base_url, cc): ❷
    url = '{}/{cc}/{cc}.gif'.format(base_url, cc=cc.lower())
    resp = yield from aiohttp.request('GET', url)
    if resp.status == 200:
        image = yield from resp.read()
        return image
    elif resp.status == 404:
        raise web.HTTPNotFound()
    else:
        raise aiohttp.HttpProcessingError(
            code=resp.status, message=resp.reason,
            headers=resp.headers)


@asyncio.coroutine
def download_one(cc, base_url, semaphore, verbose):  ❸
    try:
        with (yield from semaphore):  ❹
            image = yield from get_flag(base_url, cc)  ❺
    except web.HTTPNotFound:  ❻
        status = HTTPStatus.not_found
        msg = 'not found'
    except Exception as exc:
        raise FetchError(cc) from exc  ❼
    else:
        save_flag(image, cc.lower() + '.gif')  ❽
        status = HTTPStatus.ok
        msg = 'OK'

    if verbose and msg:
        print(cc, msg)

    return Result(status, cc)
```

❶ 这个自定义的异常用于包装其他 HTTP 或网络异常，并获取 `country_code`，以便报告错误。

❷ `get_flag` 协程有三种返回结果：返回下载得到的图像；HTTP 响应码为 404 时，抛出 `web.HTTPNotFound` 异常；返回其他 HTTP 状态码

时，抛出 aiohttp.HttpProcessingError 异常。

❸ semaphore 参数是 asyncio.Semaphore 类（https://docs.python.org/3/library/asyncio-sync.html#asyncio.Semaphore）的实例。Semaphore 类是同步装置，用于限制并发请求数量。

❹ 在 yield from 表达式中把 semaphore 当成上下文管理器使用，防止阻塞整个系统：如果 semaphore 计数器的值是所允许的最大值，只有这个协程会阻塞。

❺ 退出这个 with 语句后，semaphore 计数器的值会递减，解除阻塞可能在等待同一个 semaphore 对象的其他协程实例。

❻ 如果没找到国旗，相应地设置 Result 的状态。

❼ 其他异常当作 FetchError 抛出，传入国家代码，并使用“PEP 3134 — Exception Chaining and Embedded Tracebacks”（https://www.python.org/dev/peps/pep-3134/）引入的 raise X from Y 句法链接原来的异常。

❽ 这个函数的作用是把国旗文件保存到硬盘中。

可以看出，与依序下载版相比，示例 18-7 中的 get_flag 和 download_one 函数改动幅度很大，因为现在这两个函数是协程，要使用 yield from 做异步调用。

对于我们分析的这种网络客户端代码来说，一定要使用某种限流机制，防止向服务器发起太多并发请求，因为如果服务器过载，那么系统的整体性能可能会下降。flags2_threadpool.py 脚本（见示例 17-14）限流的方法是，在 download_many 函数中实例化 ThreadPoolExecutor 类时把 max_workers 参数的值设为 concur_req，只在线程池中启动 concur_req 个线程。在 flags2_asyncio.py 脚本中我的做法是，在 downloader_coro 函数中创建一个 asyncio.Semaphore 实例（在后面的示例 18-8 中），然后把它传给示例 18-7 中 download_one 函数的 semaphore 参数。[7]

[7]感谢 Guto Maia 指出本书的草稿没有说明 Semaphore 类。

Semaphore 对象维护着一个内部计数器，若在对象上调用 .acquire() 协程方法，计数器则递减；若在对象上调用 .release() 协程方法，计数器则递增。计数器的初始值在实例化 Semaphore 时设定，如 downloader_coro 函数中的这一行所示：

```
semaphore = asyncio.Semaphore(concur_req)
```

如果计数器大于零，那么调用 .acquire() 方法不会阻塞；可是，如果计数器为零，那么 .acquire() 方法会阻塞调用这个方法的协程，直到其他协程在同一个 Semaphore 对象上调用 .release() 方法，让计数器递增。在示例 18-7 中，我没有调用 .acquire() 或 .release() 方法，而是在 download_one 函数中的下述代码块中把 semaphore 当作上下文管理器使用：

```
        with (yield from semaphore):
            image = yield from get_flag(base_url, cc)
```

这段代码保证，任何时候都不会有超过 concur_req 个 get_flag 协程启动。

现在来分析示例 18-8 中这个脚本余下的代码。注意，download_many 函数中以前的大多数功能现在都在 downloader_coro 协程中。我们必须这么做，因为必须使用 yield from 获取 asyncio.as_completed 函数产出的期物的结果，所以 as_completed 函数必须在协程中调用。可是，我不能直接把 download_many 函数改成协程，因为必须在脚本的最后一行把 download_many 函数传给 flags2_common 模块中定义的 main 函数，可 main 函数的参数不是协程，而是一个普通的函数。因此，我定义了 downloader_coro 协程，让它运行 as_completed 循环。现在，download_many 函数只用于设置事件循环，并把 downloader_coro 协程传给 loop.run_until_complete 方法，调度 downloader_coro。

示例 **18-8**　flags2_asyncio.py：接续示例 18-7

```
@asyncio.coroutine
```

```
def downloader_coro(cc_list, base_url, verbose, concur_req):  ❶
    counter = collections.Counter()
    semaphore = asyncio.Semaphore(concur_req)  ❷
    to_do = [download_one(cc, base_url, semaphore, verbose)
             for cc in sorted(cc_list)]  ❸

    to_do_iter = asyncio.as_completed(to_do)  ❹
    if not verbose:
        to_do_iter = tqdm.tqdm(to_do_iter, total=len(cc_list))  ❺
    for future in to_do_iter:  ❻
        try:
            res = yield from future  ❼
        except FetchError as exc:  ❽
            country_code = exc.country_code  ❾
            try:
                error_msg = exc.__cause__.args[0]  ❿
            except IndexError:
                error_msg = exc.__cause__.__class__.__name__  ⓫
            if verbose and error_msg:
                msg = '*** Error for {}: {}'
                print(msg.format(country_code, error_msg))
            status = HTTPStatus.error
        else:
            status = res.status

        counter[status] += 1  ⓬

    return counter  ⓭


def download_many(cc_list, base_url, verbose, concur_req):
    loop = asyncio.get_event_loop()
    coro = downloader_coro(cc_list, base_url, verbose, concur_req)
    counts = loop.run_until_complete(coro)  ⓮
    loop.close()  ⓯
    return counts

if __name__ == '__main__':
    main(download_many, DEFAULT_CONCUR_REQ, MAX_CONCUR_REQ)
```

❶ 这个协程的参数与 download_many 函数一样，但是不能直接调用，因为它是协程函数，而不是像 download_many 那样的普通函数。

❷ 创建一个 asyncio.Semaphore 实例，最多允许激活 concur_req 个使用这个计数器的协程。

❸ 多次调用 download_one 协程，创建一个协程对象列表。

❹ 获取一个迭代器，这个迭代器会在期物运行结束后返回期物。

❺ 把迭代器传给 tqdm 函数，显示进度。

❻ 迭代运行结束的期物；这个循环与示例 17-14 中 download_many 函数里的那个十分相似；不同的部分主要是异常处理，因为两个 HTTP 库（requests 和 aiohttp）之间有差异。

❼ 获取 asyncio.Future 对象的结果，最简单的方法是使用 yield from，而不是调用 future.result() 方法。

❽ download_one 函数抛出的各个异常都包装在 FetchError 对象里，并且链接原来的异常。

❾ 从 FetchError 异常中获取错误发生时的国家代码。

❿ 尝试从原来的异常（__cause__）中获取错误消息。

⓫ 如果在原来的异常中找不到错误消息，使用所链接异常的类名作为错误消息。

⓬ 记录结果。

⓭ 与其他脚本一样，返回计数器。

⓮ download_many 函数只是实例化 downloader_coro 协程，然后通过 run_until_complete 方法把它传给事件循环。

⓯ 所有工作做完后，关闭事件循环，返回 counts。

在示例 18-8 中不能像示例 17-14 那样把期物映射到国家代码上，因为 asyncio.as_completed 函数返回的期物与传给 as_completed 函数的期物可能不同。在 asyncio 包内部，我们提供的期物会被替换成生成相同结果的期物。[8]

> [8]关于这一点的详细讨论，可以阅读我在 python-tulip 讨论组中发起的话题，题为“Which other futures my come out of asyncio.as_completed?”（https://groups.google.com/forum/#!msg/python-

> tulip/PdAEtwpaJHs/7fqb-Qj2zJoJ）。Guido 回复了，而且深入分析了 as_completed 函数的实现，还说明了 asyncio 包中期物与协程之间的紧密关系。

因为失败时不能以期物为键从字典中获取国家代码，所以我实现了自定义的 FetchError 异常（如示例 18-7 所示）。FetchError 包装网络异常，并关联相应的国家代码，因此在详细模式中报告错误时能显示国家代码。如果没有错误，那么国家代码是 for 循环顶部那个 yield from future 表达式的结果。

我们使用 asyncio 包实现的这个示例与前面的 flags2_threadpool.py 脚本具有相同的功能，这一话题到此结束。接下来，我们要改进 flags2_asyncio.py 脚本，进一步探索 asyncio 包。

在分析示例 18-7 的过程中，我发现 save_flag 函数会执行硬盘 I/O 操作，而这应该异步执行。下一节说明做法。

18.4.2　使用**Executor**对象，防止阻塞事件循环

Python 社区往往会忽略一个事实——访问本地文件系统会阻塞，想当然地认为这种操作不会受网络访问的高延迟影响（这也极难预料）。与之相比，Node.js 程序员则始终谨记，所有文件系统函数都会阻塞，因为这些函数的签名中指明了要有回调。表 18-1 已经指出，硬盘 I/O 阻塞会浪费几百万个 CPU 周期，而这可能会对应用程序的性能产生重大影响。

在示例 18-7 中，阻塞型函数是 save_flag。在这个脚本的线程版中（见示例 17-14），save_flag 函数会阻塞运行 download_one 函数的线程，但是阻塞的只是众多工作线程中的一个。阻塞型 I/O 调用在背后会释放 GIL，因此另一个线程可以继续。但是在 flags2_asyncio.py 脚本中，save_flag 函数阻塞了客户代码与 asyncio 事件循环共用的唯一线程，因此保存文件时，整个应用程序都会冻结。这个问题的解决方法是，使用事件循环对象的 run_in_executor 方法。

asyncio 的事件循环在背后维护着一个 ThreadPoolExecutor 对象，我们可以调用 run_in_executor 方法，把可调用的对象发给它执行。若想在这个示例中使用这个功能，download_one 协程只有几行代码需要改动，如示例 18-9 所示。

示例 **18-9**　flags2_asyncio_executor.py：使用默认的 `ThreadPoolExecutor` 对象运行 `save_flag` 函数

```
@asyncio.coroutine
def download_one(cc, base_url, semaphore, verbose):
    try:
        with (yield from semaphore):
            image = yield from get_flag(base_url, cc)
    except web.HTTPNotFound:
        status = HTTPStatus.not_found
        msg = 'not found'
    except Exception as exc:
        raise FetchError(cc) from exc
    else:
        loop = asyncio.get_event_loop()  ❶
        loop.run_in_executor(None,  ❷
                save_flag, image, cc.lower() + '.gif')  ❸
        status = HTTPStatus.ok
        msg = 'OK'

    if verbose and msg:
        print(cc, msg)

    return Result(status, cc)
```

❶ 获取事件循环对象的引用。

❷ `run_in_executor` 方法的第一个参数是 `Executor` 实例；如果设为 `None`，使用事件循环的默认 `ThreadPoolExecutor` 实例。

❸ 余下的参数是可调用的对象，以及可调用对象的位置参数。

我测试示例 18-9 时，没有发现改用 `run_in_executor` 方法保存图像文件后性能有明显变化，因为图像都不大（平均 13KB）。不过，如果编辑 flags2_common.py 脚本中的 `save_flag` 函数，把各个文件保存的字节数变成原来的 10 倍（只需把 `fp.write(img)` 改成 `fp.write(img*10)`），此时便会看到效果。下载的平均字节数变成 130KB 后，使用 `run_in_executor` 方法的好处就体现出来了。如果下载包含百万像素的图像，速度提升更明显。

如果需要协调异步请求，而不只是发起完全独立的请求，协程较之回调的好处会变得显而易见。下一节说明回调的问题，并给出解决方法。

18.5　从回调到期物和协程

使用协程做面向事件编程，需要下一番功夫才能掌握，因此最好知道，与经典的回调式编程相比，协程有哪些改进。这就是本节的话题。

只要对回调式面向事件编程有一定的经验，就知道“回调地狱”这个术语：如果一个操作需要依赖之前操作的结果，那就得嵌套回调。如果要连续做 3 次异步调用，那就需要嵌套 3 层回调。示例 18-10 是一个使用 JavaScript 编写的例子。

示例 **18-10**　JavaScript 中的回调地狱：嵌套匿名函数，也称为灾难金字塔

```
api_call1(request1, function (response1) {
    // 第一步
    var request2 = step1(response1);

    api_call2(request2, function (response2) {
        // 第二步
        var request3 = step2(response2);

        api_call3(request3, function (response3) {
            // 第三步
            step3(response3);
        });
    });
});
```

在示例 18-10 中，`api_call1`、`api_call2` 和 `api_call3` 是库函数，用于异步获取结果。例如，`api_call1` 从数据库中获取结果，`api_call2` 从 Web 服务器中获取结果。这 3 个函数都有回调。在 JavaScript 中，回调通常使用匿名函数实现（在下述 Python 示例中分别把这 3 个回调命名为 `stage1`、`stage2` 和 `stage3`）。这里的 `step1`、`step2` 和 `step3` 是应用程序中的常规函数，用于处理回调接收到的响应。

示例 18-11 展示 Python 中的回调地狱是什么样子。

示例 **18-11**　Python 中的回调地狱：链式回调

```
def stage1(response1):
    request2 = step1(response1)
    api_call2(request2, stage2)


def stage2(response2):
    request3 = step2(response2)
    api_call3(request3, stage3)


def stage3(response3):
    step3(response3)


api_call1(request1, stage1)
```

虽然示例 18-11 中的代码与示例 18-10 的排布方式差异很大，但是作用却完全相同。前述 JavaScript 示例也能改写成这种排布方式（但是这段 Python 代码不能改写成 JavaScript 那种风格，因为 `lambda` 表达式句法上有限制）。

示例 18-10 和示例 18-11 组织代码的方式导致代码难以阅读，也更难编写：每个函数做一部分工作，设置下一个回调，然后返回，让事件循环继续运行。这样，所有本地的上下文都会丢失。执行下一个回调时（例如 `stage2`），就无法获取 `request2` 的值。如果需要那个值，那就必须依靠闭包，或者把它存储在外部数据结构中，以便在处理过程的不同阶段使用。

在这个问题上，协程能发挥很大的作用。在协程中，如果要连续执行 3 个异步操作，只需使用 `yield3` 次，让事件循环继续运行。准备好结果后，调用 `.send()` 方法，激活协程。对事件循环来说，这种做法与调用回调类似。但是对使用协程式异步 API 的用户来说，情况就大为不同了：3 次操作都在同一个函数定义体中，像是顺序代码，能在处理过程中使用局部变量保留整个任务的上下文。请看示例 18-12。

示例 **18-12**　使用协程和 `yield from` 结构做异步编程，无需使用回调

```
@asyncio.coroutine
def three_stages(request1):
    response1 = yield from api_call1(request1)
    # 第一步
    request2 = step1(response1)
    response2 = yield from api_call2(request2)
    # 第二步
    request3 = step2(response2)
    response3 = yield from api_call3(request3)
    # 第三步
    step3(response3)


loop.create_task(three_stages(request1))  # 必须显式调度执行
```

与前面的 JavaScript 和 Python 示例相比，示例 18-12 容易理解多了：操作的 3 个步骤依次写在同一个函数中。这样，后续处理便于使用前一步的结果；而且提供了上下文，能通过异常来报告错误。

假设在示例 18-11 中处理 `api_call2(request2, stage2)` 调用（`stage1` 函数最后一行）时抛出了 I/O 异常，这个异常无法在 `stage1` 函数中捕获，因为 `api_call2` 是异步调用，还未执行任何 I/O 操作就会立即返回。在基于回调的 API 中，这个问题的解决方法是为每个异步调用注册两个回调，一个用于处理操作成功时返回的结果，另一个用于处理错误。一旦涉及错误处理，回调地狱的危害程度就会迅速增大。

与此相比，在示例 18-12 中，那个三步操作的所有异步调用都在同一个函数中（`three_stages`），如果异步调用 `api_call1`、`api_call2` 和 `api_call3` 会抛出异常，那么可以把相应的 `yield from` 表达式放在 `try/except` 块中处理异常。

这么做比陷入回调地狱好多了，但是我不会把这种方式称为协程天堂，毕竟我们还要付出代价。我们不能使用常规的函数，必须使用协程，而且要习惯 `yield from`——这是第一个障碍。只要函数中有 `yield from`，函数就会变成协程，而协程不能直接调用，即不能像示例 18-11 中那样调用 `api_call1(request1, stage1)` 来启动回调链。我们必须使用事件循环显式排定协程的执行时间，或者在其他排定了执行时间的协程中使用 `yield from` 表达式把它激活。如果示例 18-12 没有最后一行（`loop.create_task(three_stages(request1))`），那么什么

也不会发生。

下面举个例子来实践这个理论。

每次下载发起多次请求

假设保存每面国旗时，我们不仅想在文件名中使用国家代码，还想加上国家名称。那么，下载每面国旗时要发起两个请求：一个请求用于获取国旗，另一个请求用于获取图像所在目录里的 metadata.json 文件（记录着国家名称）。

在同一个任务中发起多个请求，这对线程版脚本来说很容易：只需接连发起两次请求，阻塞线程两次，把国家代码和国家名称保存在局部变量中，在保存文件时使用。如果想在异步脚本中使用回调做到这一点，你会闻到回调地狱中飘来的硫磺味道：国家代码和名称要放在闭包中传来传去，或者保存在某个地方，在保存文件时使用，这么做是因为各个回调在不同的局部上下文中运行。协程和 `yield from` 结构能缓解这个问题。解决方法虽然没有使用多个线程那么简单，但是比链式或嵌套回调易于管理。

示例 18-13 是使用 `asyncio` 包下载国旗脚本的第 3 版，这一次国旗的文件名中有国家名称。 flags2_asyncio.py 脚本（示例 18-7 和示例 18-8）中的 `download_many` 函数和 `downloader_coro` 协程没变，有变化的是下面的内容。

`download_one`

　　现在，这个协程使用 `yield from` 把职责委托给 `get_flag` 协程和新添的 `get_country` 协程。

`get_flag`

　　这个协程的大多数代码移到新添的 `http_get` 协程中了，以便也能在 `get_country` 协程中使用。

`get_country`

　　这个协程获取国家代码相应的 metadata.json 文件，从文件中读取国

家名称。

`http_get`

从 Web 获取文件的通用代码。

示例 **18-13** flags3_asyncio.py：再定义几个协程，把职责委托出去，每次下载国旗时发起两次请求

```
@asyncio.coroutine
def http_get(url):
    res = yield from aiohttp.request('GET', url)
    if res.status == 200:
        ctype = res.headers.get('Content-type', '').lower()
        if 'json' in ctype or url.endswith('json'):
            data = yield from res.json()  ❶
        else:
            data = yield from res.read()  ❷
        return data

    elif res.status == 404:
        raise web.HTTPNotFound()
    else:
        raise aiohttp.errors.HttpProcessingError(
            code=res.status, message=res.reason,
            headers=res.headers)


@asyncio.coroutine
def get_country(base_url, cc):
    url = '{}/{cc}/metadata.json'.format(base_url, cc=cc.lower())
    metadata = yield from http_get(url)  ❸
    return metadata['country']


@asyncio.coroutine
def get_flag(base_url, cc):
    url = '{}/{cc}/{cc}.gif'.format(base_url, cc=cc.lower())
    return (yield from http_get(url))  ❹


@asyncio.coroutine
def download_one(cc, base_url, semaphore, verbose):
    try:
        with (yield from semaphore):  ❺
```

```
            image = yield from get_flag(base_url, cc)
        with (yield from semaphore):
            country = yield from get_country(base_url, cc)
    except web.HTTPNotFound:
        status = HTTPStatus.not_found
        msg = 'not found'
    except Exception as exc:
        raise FetchError(cc) from exc
    else:
        country = country.replace(' ', '_')
        filename = '{}-{}.gif'.format(country, cc)
        loop = asyncio.get_event_loop()
        loop.run_in_executor(None, save_flag, image, filename)
        status = HTTPStatus.ok
        msg = 'OK'

    if verbose and msg:
        print(cc, msg)

    return Result(status, cc)
```

❶ 如果内容类型中包含 `'json'`，或者 `url` 以 `.json` 结尾，那么在响应上调用 `.json()` 方法，解析响应，返回一个 Python 数据结构——在这里是一个字典。

❷ 否则，使用 `.read()` 方法读取原始字节。

❸ `metadata` 变量的值是一个由 JSON 内容构建的 Python 字典。

❹ 这里必须在外层加上括号，如果直接写 `return yield from`，Python 解析器会不明所以，报告句法错误。

❺ 我分别在 `semaphore` 控制的两个 `with` 块中调用 `get_flag` 和 `get_country`，因为我想尽量缩减下载时间。

在示例 18-13 中，`yield from` 句法出现了 9 次。现在，你应该已经熟知如何在协程中使用这个结构把职责委托给另一个协程，而不阻塞事件循环。

问题的关键是，知道何时该使用 `yield from`，何时不该使用。基本原则很简单，`yield from` 只能用于协程和 `asyncio.Future` 实例（包括

`Task` 实例）。可是有些 API 很棘手，肆意混淆协程和普通的函数，例如下一节实现某个服务器时使用的 `StreamWriter` 类。

示例 18-13 是本书最后一次讨论 `flags2` 系列示例。我建议你自己运行那些示例，有助于对 HTTP 并发客户端的运作方式建立直观认识。你可以使用 `-a`、`-e` 和 `-l` 这三个命令行选项控制下载的国旗数量，还可以使用 `-m` 选项设置并发下载数。此外，还可以分别使用 `LOCAL`、`REMOTE`、`DELAY` 和 `ERROR` 服务器测试，找出能最大限度地利用各个服务器的吞吐量的并发下载数。如果想去掉错误或延迟，可以修改 vaurien_error_delay.sh 脚本（https://github.com/fluentpython/example-code/blob/master/17-futures/countries/vaurien_error_delay.sh）中的设置。

客户端脚本到此结束，接下来使用 `asyncio` 包编写服务器。

18.6　使用asyncio包编写服务器

演示 TCP 服务器时通常使用回显服务器。我们要构建更好玩一点的示例服务器，用于查找 Unicode 字符，分别使用简单的 TCP 协议和 HTTP 协议实现。这两个服务器的作用是，让客户端使用 4.8 节讨论过的 `unicodedata` 模块，通过规范名称查找 Unicode 字符。图 18-2 展示了在一个 Telnet 会话中访问 TCP 版字符查找服务器所做的两次查询，一次查询国际象棋棋子字符，一次查询名称中包含“sun”的字符。

```
lontra:charfinder luciano$ telnet localhost 2323
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
?> chess black
U+265A	♚	BLACK CHESS KING
U+265B	♛	BLACK CHESS QUEEN
U+265C	♜	BLACK CHESS ROOK
U+265D	♝	BLACK CHESS BISHOP
U+265E	♞	BLACK CHESS KNIGHT
U+265F	♟	BLACK CHESS PAWN
6 matches for 'chess black'
?> sun
U+2600	☀	BLACK SUN WITH RAYS
U+2609	☉	SUN
U+263C	☼	WHITE SUN WITH RAYS
U+26C5	⛅	SUN BEHIND CLOUD
U+2E9C	⺜	CJK RADICAL SUN
U+2F47	⽇	KANGXI RADICAL SUN
U+3230	㈰	PARENTHESIZED IDEOGRAPH SUN
U+3290	㊐	CIRCLED IDEOGRAPH SUN
U+C21C	순	HANGUL SYLLABLE SUN
U+1F31E	🌞	SUN WITH FACE
10 matches for 'sun'
?> ^C
Connection closed by foreign host.
lontra:charfinder luciano$
```

图 **18-2**：在一个 **Telnet** 会话中访问 **tcp_charfinder.py** 服务器——查询**“chess black”**和**“sun”**

接下来讨论实现方式。

18.6.1　使用asyncio包编写TCP服务器

下面几个示例的大多数逻辑在 charfinder.py 模块中，这个模块没有任何并发。你可以在命令行中使用 charfinder.py 脚本查找字符，不过这个脚本更为重要的作用是为使用 `asyncio` 包编写的服务器提供支持。charfinder.py 脚本的代码在本书的代码仓库中（https://github.com/fluentpython/example-code）。

`charfinder` 模块读取 Python 内建的 Unicode 数据库，为每个字符名称中的每个单词建立索引，然后倒排索引，存进一个字典。例如，在倒排索引中，`'SUN'` 键对应的条目是一个集合（`set`），里面是名称中包含 `'SUN'` 这个词的 10 个 Unicode 字符。[9] 倒排索引保存在本地一个名为 charfinder_index.pickle 的文件中。如果查询多个单词，`charfinder` 会计算从索引中所得集合的交集。

> [9]在 Python 3.5 中，新增了 4 个名称中包含 `'SUN'` 的 Unicode 字符：U+1F323（WHITE SUN）、U+1F324（WHITE SUN WITH SMALL CLOUD）、U+1F325（WHITE SUN BEHIND CLOUD）和 U+1F326（WHITE SUN BEHIND CLOUD WITH RAIN）。——编者注

下面我们把注意力集中在响应图 18-2 中那两个查询的 tcp_charfinder.py 脚本上。我要对这个脚本中的代码做大量说明，因此把它分为两部分，分别在示例 18-14 和示例 18-15 中列出。

示例 **18-14**　tcp_charfinder.py：使用 `asyncio.start_server` 函数实现的简易 TCP 服务器；这个模块余下的代码在示例 18-15 中

```
import sys
import asyncio

from charfinder import UnicodeNameIndex  ❶

CRLF = b'\r\n'
PROMPT = b'?> '

index = UnicodeNameIndex()  ❷

@asyncio.coroutine
def handle_queries(reader, writer):  ❸
    while True:  ❹
        writer.write(PROMPT)  # 不能使用yield from!  ❺
        yield from writer.drain()  # 必须使用yield from!  ❻
        data = yield from reader.readline()  ❼
        try:
```

```
            query = data.decode().strip()
        except UnicodeDecodeError:  ❽
            query = '\x00'
        client = writer.get_extra_info('peername')  ❾
        print('Received from {}: {!r}'.format(client, query))  ❿
        if query:
            if ord(query[:1]) < 32:  ⓫
                break
            lines = list(index.find_description_strs(query))  ⓬
            if lines:
                writer.writelines(line.encode() + CRLF for line in lines)  ⓭
            writer.write(index.status(query, len(lines)).encode() + CRLF)  ⓮

            yield from writer.drain()  ⓯
            print('Sent {} results'.format(len(lines)))  ⓰

    print('Close the client socket')  ⓱
    writer.close()  ⓲
```

❶ UnicodeNameIndex 类用于构建名称索引，提供查询方法。

❷ 实例化 UnicodeNameIndex 类时，它会使用 charfinder_index.pickle 文件（如果有的话），或者构建这个文件，因此第一次运行时可能要等几秒钟服务器才能启动。[10]

> [10]Leonardo Rochael 指出，可以在示例 18-15 中的 main 函数里使用 loop.run_with_executor() 方法，在另一个线程中构建 Unicode 名称索引，这样索引构建好之后，服务器能立即开始接收请求。他说得对，不过这个应用的唯一用途是查询索引，因此那样做没有多大好处。不过，Leo 建议的做法是个不错的练习，有兴趣的话你可以去做。

❸ 这个协程要传给 asyncio.start_server 函数，接收的两个参数是 asyncio.StreamReader 对象和 asyncio.StreamWriter 对象。

❹ 这个循环处理会话，直到从客户端收到控制字符后退出。

❺ StreamWriter.write 方法不是协程，只是普通的函数；这一行代码发送 ?> 提示符。

❻ StreamWriter.drain 方法刷新 writer 缓冲；因为它是协程，所以必须使用 yield from 调用。

❼ `StreamReader.readline` 方法是协程，返回一个 `bytes` 对象。

❽ Telnet 客户端发送控制字符时，可能会抛出 `UnicodeDecodeError` 异常；遇到这种情况时，为了简单起见，假装发送的是空字符。

❾ 返回与套接字连接的远程地址。

❿ 在服务器的控制台中记录查询。

⓫ 如果收到控制字符或者空字符，退出循环。

⓬ 返回一个生成器，产出包含 Unicode 码位、真正的字符和字符名称的字符串（例如，`U+0039\t9\tDIGIT NINE`）；为了简单起见，我从中构建了一个列表。

⓭ 使用默认的 UTF-8 编码把 `lines` 转换成 `bytes` 对象，并在每一行末尾添加回车符和换行符；注意，参数是一个生成器表达式。

⓮ 输出状态，例如 `627 matches for 'digit'`。[11]

> [11]在 Python 3.5 中，是 `755 matches for 'digit'`。——编者注

⓯ 刷新输出缓冲。

⓰ 在服务器的控制台中记录响应。

⓱ 在服务器的控制台中记录会话结束。

⓲ 关闭 `StreamWriter` 流。

`handle_queries` 协程的名称是复数，因为它启动交互式会话后能处理各个客户端发来的多次请求。

注意，示例 18-14 中所有的 I/O 操作都使用 `bytes` 格式。因此，我们要解码从网络中收到的字符串，还要编码发出的字符串。Python 3 默认使用的编码是 UTF-8，这里就隐式使用了这个编码。

注意一点，有些 I/O 方法是协程，必须由 `yield from` 驱动，而另一些

则是普通的函数。例如，StreamWriter.write 是普通的函数，我们假定它大多数时候都不会阻塞，因为它把数据写入缓冲；而刷新缓冲并真正执行 I/O 操作的 StreamWriter.drain 是协程，StreamReader.readline 也是协程。写作本书时，asyncio 包的 API 文档有重大的改进，明确标识出了哪些方法是协程。

示例 18-15 接续示例 18-14，列出这个模块的 main 函数。

示例 **18-15** tcp_charfinder.py（接续示例 18-14）：main 函数创建并销毁事件循环和套接字服务器

```
def main(address='127.0.0.1', port=2323):  ❶
    port = int(port)
    loop = asyncio.get_event_loop()
    server_coro = asyncio.start_server(handle_queries, address, port,
                                loop=loop)  ❷
    server = loop.run_until_complete(server_coro)  ❸
    host = server.sockets[0].getsockname()  ❹
    print('Serving on {}. Hit CTRL-C to stop.'.format(host))  ❺
    try:
        loop.run_forever()  ❻
    except KeyboardInterrupt:  # 按CTRL-C键
        pass

    print('Server shutting down.')
    server.close()  ❼
    loop.run_until_complete(server.wait_closed())  ❽
    loop.close()  ❾


if __name__ == '__main__':
    main(*sys.argv[1:]) ❿
```

❶ 调用 main 函数时可以不传入参数。

❷ asyncio.start_server 协程运行结束后，返回的协程对象返回一个 asyncio.Server 实例，即一个 TCP 套接字服务器。

❸ 驱动 server_coro 协程，启动服务器（server）。

❹ 获取这个服务器的第一个套接字的地址和端口，然后……

❺ ……在服务器的控制台中显示出来。这是这个脚本在服务器的控制台中显示的第一个输出。

❻ 运行事件循环；main 函数在这里阻塞，直到在服务器的控制台中按 CTRL-C 键才会关闭。

❼ 关闭服务器。

❽ server.wait_closed() 方法返回一个期物；调用 loop.run_until_complete 方法，运行期物。

❾ 终止事件循环。

❿ 这是处理可选的命令行参数的简便方式：展开 sys.argv[1:]，传给 main 函数，未指定的参数使用相应的默认值。

注意，run_until_complete 方法的参数是一个协程（start_server 方法返回的结果）或一个 Future 对象（server.wait_closed 方法返回的结果）。如果传给 run_until_complete 方法的参数是协程，会把协程包装在 Task 对象中。

仔细查看 tcp_charfinder.py 脚本在服务器控制台中生成的输出（如示例 18-16），更易于理解脚本中控制权的流动。

示例 **18-16** tcp_charfinder.py：这是图 18-2 所示会话在服务器端的输出

```
$ python3 tcp_charfinder.py
Serving on ('127.0.0.1', 2323). Hit CTRL-C to stop. ❶
Received from ('127.0.0.1', 62910): 'chess black' ❷
Sent 6 results
Received from ('127.0.0.1', 62910): 'sun' ❸
Sent 10 results
Received from ('127.0.0.1', 62910): '\x00'  ❹
Close the client socket ❺
```

❶ 这是 main 函数的输出。

❷ handle_queries 协程中那个 while 循环第一次迭代的输出。

❸ 那个 while 循环第二次迭代的输出。[12]

> [12]在 Python 3.5 中是 Sent 14 results。参见本小节开头的编者注。——编者注

❹ 用户按下 CTRL-C 键；服务器收到控制字符，关闭会话。

❺ 客户端套接字关闭了，但是服务器仍在运行，准备为其他客户端提供服务。

注意，main 函数几乎会立即显示 Serving on... 消息，然后在调用 loop.run_forever() 方法时阻塞。在那一点，控制权流动到事件循环中，而且一直待在那里，不过偶尔会回到 handle_queries 协程，这个协程需要等待网络发送或接收数据时，控制权又交还事件循环。在事件循环运行期间，只要有新客户端连接服务器就会启动一个 handle_queries 协程实例。因此，这个简单的服务器可以并发处理多个客户端。出现 KeyboardInterrupt 异常，或者操作系统把进程杀死，服务器会关闭。

tcp_charfinder.py 脚本利用 asyncio 包提供的高层流 API（https://docs.python.org/3/library/asyncio-stream.html），有现成的服务器可用，所以我们只需实现一个处理程序（普通的回调或协程）。此外，asyncio 包受 Twisted 框架中抽象的传送和协议启发，还提供了低层传送和协议 API。详情请参见 asyncio 包的文档（https://docs.python.org/3/library/asyncio-protocol.html），里面有一个使用低层 API 实现的 TCP 回显服务器。

下一节实现 HTTP 版字符查找服务器。

18.6.2　使用aiohttp包编写Web服务器

asyncio 版国旗下载示例使用的 aiohttp 库也支持服务器端 HTTP，我就使用这个库实现了 http_charfinder.py 脚本。图 18-3 是这个简易服务器的 Web 界面，显示搜索“cat face”表情符号得到的结果。

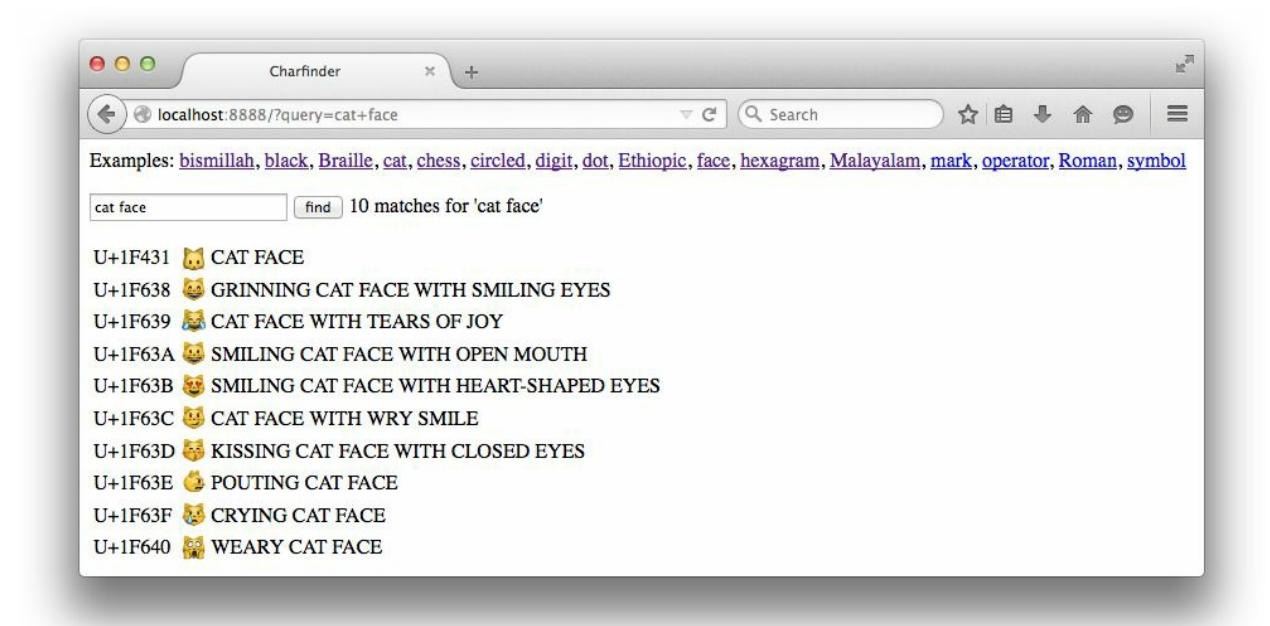


图 **18-3**：浏览器窗口中显示在 **http_charfinder.py** 服务器中搜索**“cat face”**得到的结果

> 有些浏览器显示 Unicode 字符的效果比其他浏览器好。图 18-3 中的截图在 OS X 版 Firefox 浏览器中截取，我在 Safari 中也得到了相同的结果。但是，运行在同一台设备中的最新版 Chrome 和 Opera 却不能显示猫脸等表情符号。不过其他搜索结果（例如“chess”）正常，因此这可能是 OS X 版 Chrome 和 Opera 的字体问题。

我们先分析 http_charfinder.py 脚本中最重要的后半部分：启动和关闭事件循环与 HTTP 服务器。参见示例 18-17。

示例 **18-17**　http_charfinder.py：`main` 和 `init` 函数

```
@asyncio.coroutine
def init(loop, address, port):  ❶
    app = web.Application(loop=loop)  ❷
    app.router.add_route('GET', '/', home)  ❸
    handler = app.make_handler()  ❹
    server = yield from loop.create_server(handler,
                                           address, port)  ❺
    return server.sockets[0].getsockname()  ❻
```

```
def main(address="127.0.0.1", port=8888):
    port = int(port)
    loop = asyncio.get_event_loop()
    host = loop.run_until_complete(init(loop, address, port))  ❼
    print('Serving on {}. Hit CTRL-C to stop.'.format(host))
    try:
        loop.run_forever() ❽
    except KeyboardInterrupt:  # 按CTRL-C键
        pass
    print('Server shutting down.')
    loop.close()  ❾


if __name__ == '__main__':
    main(*sys.argv[1:])
```

❶ init 协程产出一个服务器，交给事件循环驱动。

❷ aiohttp.web.Application 类表示 Web 应用……

❸ ……通过路由把 URL 模式映射到处理函数上；这里，把 GET / 路由映射到 home 函数上（参见示例 18-18）。

❹ app.make_handler 方法返回一个 aiohttp.web.RequestHandler 实例，根据 app 对象设置的路由处理 HTTP 请求。

❺ create_server 方法创建服务器，以 handler 为协议处理程序，并把服务器绑定在指定的地址（address）和端口（port）上。

❻ 返回第一个服务器套接字的地址和端口。

❼ 运行 init 函数，启动服务器，获取服务器的地址和端口。

❽ 运行事件循环；控制权在事件循环手上时，main 函数会在这里阻塞。

❾ 关闭事件循环。

我们已经熟悉了 asyncio 包的 API，现在可以对比一下示例 18-17 与前

面的 TCP 示例（见示例 18-15），看它们创建服务器的方式有何不同。

在前面的 TCP 示例中，服务器通过 main 函数中的下面两行代码创建并排定运行时间：

```
server_coro = asyncio.start_server(handle_queries, address, port,
                                   loop=loop)
server = loop.run_until_complete(server_coro)
```

在这个 HTTP 示例中，init 函数通过下述方式创建服务器：

```
server = yield from loop.create_server(handler,
                                       address, port)
```

但是 init 是协程，驱动它运行的是 main 函数中的这一行：

```
host = loop.run_until_complete(init(loop, address, port))
```

asyncio.start_server 函数和 loop.create_server 方法都是协程，返回的结果都是 asyncio.Server 对象。为了启动服务器并返回服务器的引用，这两个协程都要由他人驱动，完成运行。在 TCP 示例中，做法是调用 loop.run_until_complete(server_coro)，其中 server_coro 是 asyncio.start_server 函数返回的结果。在 HTTP 示例中，create_server 方法在 init 协程中的一个 yield from 表达式里调用，而 init 协程则由 main 函数中的 loop.run_until_complete(init(...)) 调用驱动。

我提到这一点是为了强调之前讨论过的一个基本事实：只有驱动协程，协程才能做事，而驱动 asyncio.coroutine 装饰的协程有两种方法，要么使用 yield from，要么传给 asyncio 包中某个参数为协程或期物的函数，例如 run_until_complete。

示例 18-18 列出 home 函数。根据这个 HTTP 服务器的配置，home 函数用于处理 /（根）URL。

示例 **18-18**　http_charfinder.py：`home` 函数

```
def home(request):  ❶
    query = request.GET.get('query', '').strip()  ❷
    print('Query: {!r}'.format(query))  ❸
    if query:  ❹
        descriptions = list(index.find_descriptions(query))
        res = '\n'.join(ROW_TPL.format(**vars(descr))
                        for descr in descriptions)
        msg = index.status(query, len(descriptions))
    else:
        descriptions = []
        res = ''
        msg = 'Enter words describing characters.'

    html = template.format(query=query, result=res,  ❺
                           message=msg)
    print('Sending {} results'.format(len(descriptions)))  ❻
    return web.Response(content_type=CONTENT_TYPE, text=html)  ❼
```

❶ 一个路由处理函数，参数是一个 `aiohttp.web.Request` 实例。

❷ 获取查询字符串，去掉首尾的空白。

❸ 在服务器的控制台中记录查询。

❹ 如果有查询字符串，从索引（`index`）中找到结果，使用 HTML 表格中的行渲染结果，把结果赋值给 `res` 变量，再把状态消息赋值给 `msg` 变量。

❺ 渲染 HTML 页面。

❻ 在服务器的控制台中记录响应。

❼ 构建 `Response` 对象，将其返回。

注意，`home` 不是协程，既然定义体中没有 `yield from` 表达式，也没必要是协程。在 `aiohttp` 包的文档中，`add_route` 方法的条目（http://aiohttp.readthedocs.org/en/v0.14.4/web_reference.html#aiohttp.web.U 下面说道，“如果处理程序是普通的函数，在内部会将其转换成协程”。

示例 18-18 中的 home 函数虽然简单，却有一个缺点。home 是普通的函数，而不是协程，这一事实预示着一个更大的问题：我们需要重新思考如何实现 Web 应用，以获得高并发。下面来分析这个问题。

18.6.3　更好地支持并发的智能客户端

示例 18-18 中的 home 函数很像是 Django 或 Flask 中的视图函数，实现方式完全没有考虑异步：获取请求，从数据库中读取数据，然后构建响应，渲染完整的 HTML 页面。在这个示例中，存储在内存中的 UnicodeNameIndex 对象是“数据库”。但是，对真正的数据库来说，应该异步访问，否则在等待数据库查询结果的过程中，事件循环会阻塞。例如，aiopg 包（https://aiopg.readthedocs.org/en/stable/）提供了一个异步 PostgreSQL 驱动，与 asyncio 包兼容；这个包支持使用 yield from 发送查询和获取结果，因此视图函数的表现与真正的协程一样。

除了防止阻塞调用之外，高并发的系统还必须把复杂的工作分成多步，以保持敏捷。http_charfinder.py 服务器表明了这一点：如果搜索“cjk”，得到的结果是 75 821 个中文、日文和韩文象形文字。[13] 此时，home 函数会返回一个 5.3MB 的 HTML 文档，显示一个有 75 821 行的表格。

> [13]这正是 CJK 表示的意思：不断增加的中文、日文和韩文字符。以后的 Python 版本支持的 CJK 象形文字数量可能会比 Python 3.4 多。

我在自己的设备中使用命令行 HTTP 客户端 curl 访问架设在本地的 http_charfinder.py 服务器，查询“cjk”，2 秒钟后获得响应。浏览器要布局包含这么大一个表格的页面，用的时间会更长。当然，大多数查询返回的响应要小得多：查询“braille”返回 256 行结果，页面大小为 19KB，在我的设备中用时 0.017 秒。可是，如果服务器要用 2 秒钟处理“cjk”查询，那么其他所有客户端都至少要等 2 秒——这是不可接受的。

避免响应时间太长的方法是实现分页：首次至多返回（比如说）200 行，用户点击链接或滚动页面时再获取更多结果。如果查看本书代码仓库（https://github.com/fluentpython/example-code）中的 charfinder.py 模块，你会发现 UnicodeNameIndex.find_descriptions 方法有两个可选的参数——start 和 stop，这是偏移值，用于支持分页。因此，我们可以返回前 200 个结果，当用户想查看更多结果时，再使用 AJAX 或 WebSockets 发送下一批结果。

实现分批发送结果所需的大多数代码都在浏览器这一端，因此 Google 和所有大型互联网公司都大量依赖客户端代码构建服务：智能的异步客户端能更好地使用服务器资源。

虽然智能的客户端甚至对老式 Django 应用也有帮助，但是要想真正为这种客户端服务，我们需要全方位支持异步编程的框架，从处理 HTTP 请求和响应到访问数据库，全都支持异步。如果想实现实时服务，例如游戏和以 WebSockets 支持的媒体流，那就尤其应该这么做。[14]

> [14]在“杂谈”中我会进一步说明这个趋势。

这里留一个练习给读者：改进 http_charfinder.py 脚本，添加下载进度条。此外还有一个附加题：实现 Twitter 那样的“无限滚动”。做完这个练习后，我们对如何使用 `asyncio` 包做异步编程的讨论就结束了。

18.7　本章小结

本章介绍了在 Python 中做并发编程的一种全新方式，这种方式使用 `yield from`、协程、期物和 `asyncio` 事件循环。首先，我们分析了两个简单的示例——两个旋转指针脚本，仔细对比了使用 `threading` 模块和 `asyncio` 包处理并发的异同。

然后，本章讨论了 `asyncio.Future` 类的细节，重点讲述它对 `yield from` 的支持，以及与协程和 `asyncio.Task` 类的关系。接下来分析了 `asyncio` 版国旗下载脚本。

然后，本章分析了 Ryan Dahl 对 I/O 延迟所做的统计数据，还说明了阻塞调用的影响。尽管有些函数必然会阻塞，但是为了让程序持续运行，有两种解决方案可用：使用多个线程，或者异步调用——后者以回调或协程的形式实现。

其实，异步库依赖于低层线程（直至内核级线程），但是这些库的用户无需创建线程，也无需知道用到了基础设施中的低层线程。在应用中，我们只需确保没有阻塞的代码，事件循环会在背后处理并发。异步系统能避免用户级线程的开销，这是它能比多线程系统管理更多并发连接的主要原因。

之后，我们又回到下载国旗的脚本，添加进度条并处理错误。这需要大幅度重构，特别是要把 `asyncio.wait` 函数换成 `asyncio.as_completed` 函数，因此不得不把 `download_many` 函数的大多数功能移到新添的 `downloader_coro` 协程中，这样我们才能使用 `yield from` 从 `asyncio.as_completed` 函数生成的多个期物中逐个获得结果。

然后，本章说明了如何使用 `loop.run_in_executor` 方法把阻塞的作业（例如保存文件）委托给线程池做。

接着，本章讨论了如何使用协程解决回调的主要问题：执行分成多步的异步任务时丢失上下文，以及缺少处理错误所需的上下文。

然后又举了一个例子，在下载国旗图像的同时获取国家名称，以此说明如何结合协程和 `yield from` 避免所谓的回调地狱。如果忽略 `yield from` 关键字，使用 `yield from` 结构实现异步调用的多步过程看起来类似于顺序执行的代码。

本章最后两个示例是使用 `asyncio` 包实现的 TCP 和 HTTP 服务器，用于按名称搜索 Unicode 字符。在分析 HTTP 服务器的最后，我们讨论了客户端 JavaScript 对服务器端提供高并发支持的重要性。使用 JavaScript，客户端可以按需发起小型请求，而不用下载较大的 HTML 页面。

18.8　延伸阅读

Python 核心开发者 Nick Coghlan 在 2013 年 1 月对“PEP 3156—Asynchronous IO Support Rebooted: the‘asyncio’Module”（https://www.python.org/dev/peps/pep-3156/）草案评论如下：

> 在这个 PEP 的开头部分应该言简意赅地说明等待异步期物返回结果的两个 API：
>
> (1) `f.add_done_callback(...)`
>
> (2) 协程中的 `yield from f`（期物运行结束后恢复协程，期物要么返回结果，要么抛出合适的异常）
>
> 此刻，这两个 API 深埋在众多的 API 中，而它们是理解核心事件循环层之上各种事物交互方式的关键。[15]

> [15]摘自 2013 年 1 月 20 日发布在 python-ideas 邮件列表中的一个消息（https://mail.python.org/pipermail/python-ideas/2013-January/018791.html），在这个消息中，Coghlan 对 PEP 3156 做出了上述评论。

PEP 3156 的作者 Guido van Rossum 没有采纳 Coghlan 的建议。实现 PEP 3156 的初期，`asyncio` 包的文档虽然十分详细，但对用户并不友好。`asyncio` 包的文档有 9 个 .rst 文件，128KB，将近 71 页。在标准库文档中，只有“Built-in Types”一章（https://docs.python.org/3/library/stdtypes.html）有这么长，而那一章内容众多，涵盖了数字类型、序列类型、生成器、映射、集合、`bool`、上下文管理器，等等。

`asyncio` 包的文档大部分是在讲概念和 API，其中夹杂着有用的示意图和示例，不过特别实用的一节是“18.5.11. Develop with asyncio”（https://docs.python.org/3/library/asyncio-dev.html），[16] 其中说明了极为重要的使用模式。`asyncio` 包的文档需要用更多的内容来说明如何使用 `asyncio`。

> [16]目前是：18.5.9. Develop with asyncio。——编者注

`asyncio` 包很新，已出版的书中少有涉及。我发现只有 Jan Palach 写的 *Parallel Programming with Python*（Packt 出版社，2014 年）一书中有一章讲到了 `asyncio`，可惜那一章很短。

不过，有很多关于 `asyncio` 的精彩演讲。我觉得最棒的是 Brett Slatkin 在蒙特利尔 PyCon 2014 大会上发表的演讲，题为“Fan-In and Fan-Out: The Crucial Components of Concurrency”（https://speakerdeck.com/pycon2014/fan-in-and-fan-out-the-crucial-components-of-concurrency-by-brett-slatkin），副标题是“Why do we need Tulip? (a.k.a., PEP 3156—asyncio)”（视频：https://www.youtube.com/watch?v=CWmq-jtkemY）。在 30 分钟内，Slatkin 实现了一个简单的 Web 爬虫示例，强调了 `asyncio` 包的正确用法。身为观众的 Guido van Rossum 提到，为了引荐 `asyncio` 包，他也写了一个 Web 爬虫。Guido 写的代码不依赖 `aiohttp` 包，只用到了标准库。Slatkin 还写了一篇见解深刻的文章，题为“Python's asyncio Is for Composition, Not Raw Performance”（http://www.onebigfluke.com/2015/02/asyncio-is-for-composition.html）。

Guido van Rossum 自己的几个演讲也是必看的，包括在 PyCon US 2013 上所做的主题演讲（http://pyvideo.org/video/1667/keynote-1），以及在 LinkedIn 公司（https://www.youtube.com/watch?v=aurOB4qYuFM）和 Twitter 大学（https://www.youtube.com/watch?v=1coLC-MUCJc）所做的演讲。此外，还推荐 Saúl Ibarra Corretgé的演讲——“A Deep Dive into PEP-3156 and the New asyncio Module”[（幻灯片（http://www.slideshare.net/saghul/asyncio），视频（https://www.youtube.com/watch?v=MS1L2RGKYyY）]。

在 PyCon US 2013 大会上，Dino Viehland 做了一场演讲，题为“Using futures for async GUI programming in Python 3.3”（http://pyvideo.org/video/1762/using-futures-for-async-gui-programming-in-python），说明如何把 `asyncio` 包集成到 Tkinter 事件循环中。Viehland 展示了在另一个事件循环之上实现 `asyncio.AbstractEventLoop` 接口的重要部分是多么容易。他的代码使用 Tulip 编写，这是 `asyncio` 包添加到标准库中之前的名称。我修改

了他的代码，以便支持 Python 3.4 中的 asyncio 包。我重构后的新版在 GitHub 中（https://github.com/fluentpython/asyncio-tkinter）。

Victor Stinner [asyncio 包的核心贡献者，asyncio 包的移植版 Trollius（http://trollius.readthedocs.org）的作者] 经常更新相关资源的链接列表——“The new Python asyncio module aka‘tulip’”（http://haypo-notes.readthedocs.org/asyncio.html）。此外，收集 asyncio 资源的还有 Asyncio.org 网站（http://asyncio.org/）和 GitHub 中的 aio-libs 组织（https://github.com/aio-libs），在这两个网站中能找到 PostgreSQL、MySQL 和多种 NoSQL 数据库的异步驱动。我没有测试过这些驱动，不过写作本书时，这些项目好像十分活跃。

Web 服务将成为 asyncio 包的重要使用场景。你的代码有可能要依赖 Andrew Svetlov 领衔开发的 aiohttp 库（http://aiohttp.readthedocs.org/en/）。你可能还想架设环境，测试错误处理代码，在这方面，Alexis Métaireau 和 Tarek Ziadé开发的 Vaurien（“混沌 TCP 代理”，http://vaurien.readthedocs.org/en/1.8/）极其有用。Vaurien 是为 Mozilla Services 项目（https://mozilla-services.github.io/）开发的，用于在程序与后端服务器（例如，数据库和 Web 服务提供方）之间的 TCP 流量中引入延迟和随机错误。

杂谈

至尊循环

有很长一段时间，大多数 Python 高手开发网络应用时喜欢使用异步编程，但是总会遇到一个问题——挑选的库之间不兼容。Ryan Dahl 提到，Twisted 是 Node.js 的灵感来源之一；而在 Python 中，Tornado 拥护使用协程做面向事件编程。

在 JavaScript 社区里还有争论，有些人推崇使用简单的回调，而有些人提倡使用与回调处于竞争地位的各种高层抽象方式。Node.js 早期版本的 API 使用的是 Promise 对象（类似于 Python 中的期物），但是后来 Ryan Dahl 决定统一只用回调。James Coglan 认为，Node.js 在这一点上错过了大好良机（https://blog.jcoglan.com/2013/03/30/callbacksare-imperative-promises-are-functional-nodes-biggest-missed-opportunity/）。

Python 社区的争论已经结束：asyncio 包添加到标准库中之后，协程和期物被确定为符合 Python 风格的异步代码编写方式。此外，asyncio 包为异步期物和事件循环定义了标准接口，为二者提供了实现参考。

正如“Python 之禅”所说：

> 肯定有一种——通常也是唯一一种——最佳的解决方案
>
> 不过这并不容易找到，因为你不是 Python 之父

或许变成荷兰人才能理解 yield from 吧。[17] 对我这个巴西人来说，一开始并不易于理解，不过一段时间之后我理解了。

更重要的是，设计 asyncio 包时考虑到了使用外部包替换自身的事件循环，因此才有 asyncio.get_event_loop 和 set_event_loop 函数——二者是抽象的事件循环策略 API（https://docs.python.org/3/library/asyncio-eventloops.html#event-loop-policies-and-the-default-policy）的一部分。

Tornado 已经有实现 asyncio.AbstractEventLoop 接口的类——AsyncIOMainLoop（http://tornado.readthedocs.org/en/latest/asyncio.
因此在同一个事件循环中可以使用这两个库运行异步代码。此外，Quamash 项目（https://pypi.python.org/pypi/Quamash/）也很有趣，它把 asyncio 包集成到 Qt 事件循环中，以便使用 PyQt 或 PySide 开发 GUI 应用。我只是举两个例子，说明 asyncio 包能把面向事件的包集成在一起。

智能的 HTTP 客户端，例如单页 Web 应用（如 Gmail）或智能手机应用，需要快速、轻量级的响应和推送更新。鉴于这样的需求，服务器端最好使用异步框架，不要使用传统的 Web 框架（如 Django）。传统框架的目的是渲染完整的 HTML 网页，而且不支持异步访问数据库。

WebSockets 协议的作用是为始终连接的客户端（例如游戏和流式应用）提供实时更新，因此，高并发的异步服务器要不间断地与成百上千个客户端交互。asyncio 包的架构能很好地支持 WebSockets，而且至少有两个库已经在 asyncio 包的基础上实现了 WebSockets

协议：Autobahn|Python（http://autobahn.ws/python/）和WebSockets（http://aaugustin.github.io/websockets/）。

“实时 Web”的整体发展趋势迅猛，这是 Node.js 需求量不断攀升的主要因素，也是 Python 生态系统积极向 `asyncio` 靠拢的重要原因。不过，要做的事还有很多。为了便于入门，我们要在标准库中提供异步 HTTP 服务器和客户端 API，异步数据库 API 3.0（https://www.python.org/dev/peps/pep-0249/），[18] 以及使用 `asyncio` 包构建的新数据库驱动。

与 Node.js 相比，含有 `asyncio` 包的 Python 3.4 最大的优势是 Python 本身：Python 语言设计良好，使用协程和 `yield from` 结构编写的异步代码比 JavaScript 采用的古老回调易于维护。而我们最大的劣势是库，Python 自带了很多库，但是那些库不支持异步编程。Node.js 库的生态系统丰富，完全建构在异步调用之上。但是，Python 和 Node. js 都有一个问题，而 Go 和 Erlang 从一开始就解决了这个问题：我们编写的代码无法轻松地利用所有可用的 CPU 核心。

Python 标准化了事件循环接口，还提供了一个异步库，这是一大进步，而且只有我们仁慈的独裁者能在众多深入人心且高质量的替代方案中选择这种方式。具体实现时，他咨询了多个重要的 Python 异步框架的作者，其中受 Glyph Lefkowitz（Twisted 的主要开发者）的影响最深。如果你想知道为什么 `asyncio.Future` 类与 Twisted 中的 `Deferred` 类不同，一定要阅读 Guido 在 Python-tulip 讨论组中发布的一篇文章，题为“Deconstructing Deferred”（https://groups.google.com/forum/#!msg/python-tulip/ut4vTG-08k8/PWZzUXX9HYIJ）。Guido 对 Twisted 这个最古老也是最大的 Python 异步框架充满敬意，在 python-twisted 讨论组中讨论设计方案时，他甚至说，“What Would Twisted Do（WWTD）”。[19]

幸好有 Guido van Rossum 打头阵，让 Python 以更好的姿态应对当前的并发挑战。若想精通 `asyncio` 包，一定要下一番功夫。可是，如果你计划使用 Python 编写并发网络应用，那就去寻求至尊循环（the One Loop）：

> 至尊循环驭众生，至尊循环寻众生，

至尊循环引众生，普照众生欣欣荣。

> [17]Python 之父 Guido van Rossum 是荷兰人。——译者注

> [18]应该是：PEP 249—Python Database API Specification v2.0。——编者注

> [19]出自 Guido 于 2015 年 1 月 29 日发布的消息（https://groups.google.com/forum/#!msg/python-tulip/pPMwts-CvUcw/eIoX_n8FSPwJ），然后 Glyph 立即回复了这一消息。

第六部分　元编程

第 19 章　动态属性和特性

> 特性至关重要的地方在于，特性的存在使得开发者可以非常安全并且确定可行地将公共数据属性作为类的公共接口的一部分开放出来。[1]

——Alex Martelli
Python 贡献者和图书作者

[1]《Python 技术手册（第 2 版）》第 101 页。（该书中文版把“property”译为属性，这里改为“特性”，其他内容与原来的翻译相同。——译者注）

在 Python 中，数据的属性和处理数据的方法统称属性（attribute）。其实，方法只是可调用的属性。除了这二者之外，我们还可以创建特性（property），在不改变类接口的前提下，使用存取方法（即读值方法和设值方法）修改数据属性。这与统一访问原则相符：

> 不管服务是由存储还是计算实现的，一个模块提供的所有服务都应该通过统一的方式使用。[2]

[2]Bertrand Meyer, *Object-Oriented Software Construction*, 2E, p. 57.

除了特性，Python 还提供了丰富的 API，用于控制属性的访问权限，以及实现动态属性。使用点号访问属性时（如 `obj.attr`），Python 解释器会调用特殊的方法（如 `__getattr__` 和 `__setattr__`）计算属性。用户自己定义的类可以通过 `__getattr__` 方法实现“虚拟属性”，当访问不存在的属性时（如 `obj.no_such_attribute`），即时计算属性的值。

动态创建属性是一种元编程，框架的作者经常这么做。然而，在 Python 中，相关的基础技术十分简单，任何人都可以使用，甚至在日常的数据转换任务中也能用到。下面以这种任务开启本章的话题。

19.1　使用动态属性转换数据

在接下来的几个示例中，我们要使用动态属性处理 O'Reilly 为 OSCON 2014 大会提供的 JSON 格式数据源。示例 19-1 是那个数据源中的 4 个记录。[3]

> [3]关于这个数据源及其使用规则，请阅读“DIY: OSCON schedule”一文（http://conferences.oreilly.com/oscon/oscon2014/public/content/schedulefeed）。那个 JSON 文件有 744KB，我写作本书时还在网上（http://www.oreilly.com/pub/sc/osconfeed）。本书代码仓库中的 oscon-schedule/data/ 目录里有个副本，文件名为 osconfeed.json（https://github.com/fluentpython/example-code/tree/master/19-dyn-attr-prop/oscon/data）。

示例 **19-1**　osconfeed.json 文件中的记录示例；节略了部分字段的内容

```
{ "Schedule":
  { "conferences": [{"serial": 115 }],
    "events": [
      { "serial": 34505,
        "name": "Why Schools Don´t Use Open Source to Teach Programming",
        "event_type": "40-minute conference session",
        "time_start": "2014-07-23 11:30:00",
        "time_stop": "2014-07-23 12:10:00",
        "venue_serial": 1462,
        "description": "Aside from the fact that high school programming...",
        "website_url": "http://oscon.com/oscon2014/public/schedule/detail/345
        "speakers": [157509],
        "categories": ["Education"] }
    ],
    "speakers": [
      { "serial": 157509,
        "name": "Robert Lefkowitz",
        "photo": null,
        "url": "http://sharewave.com/",
        "position": "CTO",
        "affiliation": "Sharewave",
        "twitter": "sharewaveteam",
        "bio": "Robert ´r0ml´ Lefkowitz is the CTO at Sharewave, a startup...
    ],
    "venues": [
      { "serial": 1462,
        "name": "F151",
```

```
        "category": "Conference Venues" }
    ]
  }
}
```

那个 JSON 源中有 895 条记录，示例 19-1 只列出了 4 条。可以看出，整个数据集是一个 JSON 对象，里面有一个键，名为 "Schedule"；这个键对应的值也是一个映像，有 4 个键：
"conferences"、"events"、"speakers" 和 "venues"。这 4 个键对应的值都是一个记录列表。在示例 19-1 中，各个列表中只有一条记录。然而，在完整的数据集中，列表中有成百上千条记录。不过，"conferences" 键对应的列表中只有一条记录，如上述示例所示。这 4 个列表中的每个元素都有一个名为 "serial" 的字段，这是元素在各个列表中的唯一标识符。

我编写的第一个脚本只用于下载那个 OSCON 数据源。为了避免浪费流量，我会先检查本地有没有副本。这么做是合理的，因为 OSCON 2014 大会已经结束，数据源不会再更新。

示例 19-2 没用到元编程，几乎所有代码的作用可以用这一个表达式概括：json.load(fp)。不过，这样足以处理那个数据集了。osconfeed.load 函数会在后面几个示例中用到。

示例 **19-2**　osconfeed.py：下载 osconfeed.json（doctest 在示例 19-3 中）

```
from urllib.request import urlopen
import warnings
import os
import json

URL = 'http://www.oreilly.com/pub/sc/osconfeed'
JSON = 'data/osconfeed.json'


def load():
    if not os.path.exists(JSON):
        msg = 'downloading {} to {}'.format(URL, JSON)
        warnings.warn(msg)  ❶
        with urlopen(URL) as remote, open(JSON, 'wb') as local:  ❷
```

```
        local.write(remote.read())

with open(JSON) as fp:
    return json.load(fp)  ❸
```

❶ 如果需要下载，就发出提醒。

❷ 在 `with` 语句中使用两个上下文管理器（从 Python 2.7 和 Python 3.1 起允许这么做），分别用于读取和保存远程文件。

❸ `json.load` 函数解析 JSON 文件，返回 Python 原生对象。在这个数据源中有这几种数据类型：`dict`、`list`、`str` 和 `int`。

有了示例 19-2 中的代码，我们可以审查数据源中的任何字段，如示例 19-3 所示。

示例 **19-3**　osconfeed.py：示例 19-2 的 doctest

```
>>> feed = load()  ❶
>>> sorted(feed['Schedule'].keys())  ❷
['conferences', 'events', 'speakers', 'venues']
>>> for key, value in sorted(feed['Schedule'].items()):
...     print('{:3} {}'.format(len(value), key))  ❸
...
  1 conferences
494 events
357 speakers
 53 venues
>>> feed['Schedule']['speakers'][-1]['name']  ❹
'Carina C. Zona'
>>> feed['Schedule']['speakers'][-1]['serial']  ❺
141590
>>> feed['Schedule']['events'][40]['name']
'There *Will* Be Bugs'
>>> feed['Schedule']['events'][40]['speakers']  ❻
[3471, 5199]
```

❶ `feed` 的值是一个字典，里面嵌套着字典和列表，存储着字符串和整数。

❷ 列出 `"Schedule"` 键中的 4 个记录集合。

❸ 显示各个集合中的记录数量。

❹ 深入嵌套的字典和列表，获取最后一个演讲者的名字。

❺ 获取那位演讲者的编号。

❻ 每个事件都有一个 'speakers' 字段，列出 0 个或多个演讲者的编号。

19.1.1　使用动态属性访问JSON类数据

示例 19-2 十分简单，不过，feed['Schedule']['events'][40]['name'] 这种句法很冗长。在 JavaScript 中，可以使用 feed.Schedule.events[40].name 获取那个值。在 Python 中，可以实现一个近似字典的类（网上有大量实现）[4]，达到同样的效果。我自己实现了 FrozenJSON 类，比大多数实现都简单，因为只支持读取，即只能访问数据。不过，这个类能递归，自动处理嵌套的映射和列表。

> [4]最常提到的一个实现是 AttrDict（https://pypi.python.org/pypi/attrdict），还有一个实现能快速创建嵌套的映射——addict（https://pypi.python.org/pypi/addict）。

示例 19-4 演示 FrozenJSON 类的用法，源代码在示例 19-5 中。

示例 **19-4**　示例 19-5 定义的 FrozenJSON 类能读取属性，如 name，还能调用方法，如 .keys() 和 .items()

```
>>> from osconfeed import load
>>> raw_feed = load()
>>> feed = FrozenJSON(raw_feed)  ❶
>>> len(feed.Schedule.speakers)  ❷
357
>>> sorted(feed.Schedule.keys())  ❸
['conferences', 'events', 'speakers', 'venues']
>>> for key, value in sorted(feed.Schedule.items()):  ❹
...     print('{:3} {}'.format(len(value), key))
...
  1 conferences
494 events
357 speakers
 53 venues
```

```
>>> feed.Schedule.speakers[-1].name  ❺
'Carina C. Zona'
>>> talk = feed.Schedule.events[40]
>>> type(talk)  ❻
<class 'explore0.FrozenJSON'>
>>> talk.name
'There *Will* Be Bugs'
>>> talk.speakers  ❼
[3471, 5199]
>>> talk.flavor  ❽
Traceback (most recent call last):
  ...
KeyError: 'flavor'
```

❶ 传入嵌套的字典和列表组成的 `raw_feed`，创建一个 `FrozenJSON` 实例。

❷ `FrozenJSON` 实例能使用属性表示法遍历嵌套的字典；这里，我们获取演讲者列表的元素数量。

❸ 也可以使用底层字典的方法，例如 `.keys()`，获取记录集合的名称。

❹ 使用 `items()` 方法获取各个记录集合及其内容，然后显示各个记录集合中的元素数量。

❺ 列表，例如 `feed.Schedule.speakers`，仍是列表；但是，如果里面的元素是映射，会转换成 `FrozenJSON` 对象。

❻ `events` 列表中的 40 号元素是一个 JSON 对象，现在则变成一个 `FrozenJSON` 实例。

❼ 事件记录中有一个 `speakers` 列表，列出演讲者的编号。

❽ 读取不存在的属性会抛出 `KeyError` 异常，而不是通常抛出的 `AttributeError` 异常。

`FrozenJSON` 类的关键是 `__getattr__` 方法。我们在 10.5 节的 `Vector` 示例中用过这个方法，那时用于通过字母获取 `Vector` 对象的分量（例如 `v.x`、`v.y`、`v.z`）。我们要记住重要的一点，仅当无法使用常规的方

式获取属性（即在实例、类或超类中找不到指定的属性），解释器才会调用特殊的 `__getattr__` 方法。

示例 19-4 的最后一行揭露了这个实现的一个小问题：理论上，尝试读取不存在的属性应该抛出 `AttributeError` 异常。其实，一开始我对这个异常做了处理，但是 `__getattr__` 方法的代码量增加了一倍，而且偏离了我最想展示的重要逻辑，因此为了教学，后来我把那部分代码去掉了。

如示例 19-5 所示，`FrozenJSON` 类只有两个方法（`__init__` 和 `__getattr__`）和一个实例属性 `__data`。因此，尝试获取其他属性会触发解释器调用 `__getattr__` 方法。这个方法首先查看 `self.__data` 字典有没有指定名称的属性（不是键），这样 `FrozenJSON` 实例便可以处理字典的所有方法，例如把 `items` 方法委托给 `self.__data.items()` 方法。如果 `self.__data` 没有指定名称的属性，那么 `__getattr__` 方法以那个名称为键，从 `self.__data` 中获取一个元素，传给 `FrozenJSON.build` 方法。这样就能深入 JSON 数据的嵌套结构，使用类方法 `build` 把每一层嵌套转换成一个 `FrozenJSON` 实例。

示例 **19-5**　explore0.py：把一个 JSON 数据集转换成一个嵌套着 `FrozenJSON` 对象、列表和简单类型的 `FrozenJSON` 对象

```
from collections import abc


class FrozenJSON:
    """一个只读接口，使用属性表示法访问JSON类对象
    """

    def __init__(self, mapping):
        self.__data = dict(mapping)  ❶

    def __getattr__(self, name):  ❷
        if hasattr(self.__data, name):
            return getattr(self.__data, name)  ❸
        else:
            return FrozenJSON.build(self.__data[name])  ❹

    @classmethod
    def build(cls, obj):  ❺
```

```
        if isinstance(obj, abc.Mapping):  ❻
            return cls(obj)
        elif isinstance(obj, abc.MutableSequence):  ❼
            return [cls.build(item) for item in obj]
        else:  ❽
            return obj
```

❶ 使用 `mapping` 参数构建一个字典。这么做有两个目的：(1) 确保传入的是字典（或者是能转换成字典的对象）；(2) 安全起见，创建一个副本。

❷ 仅当没有指定名称（`name`）的属性时才调用 `__getattr__` 方法。

❸ 如果 `name` 是实例属性 `__data` 的属性，返回那个属性。调用 `keys` 等方法就是通过这种方式处理的。

❹ 否则，从 `self.__data` 中获取 `name` 键对应的元素，返回调用 `FrozenJSON.build()` 方法得到的结果。[5]

> [5]这一行中的 `self.__data[name]` 表达式可能抛出 `KeyError` 异常。我们应该处理这个异常，抛出 `AttributeError` 异常，因为这才是 `__getattr__` 方法应该抛出的异常种类。建议勤奋的读者实现错误处理代码，当作一个练习。

❺ 这是一个备选构造方法，`@classmethod` 装饰器经常这么用。

❻ 如果 `obj` 是映射，那就构建一个 `FrozenJSON` 对象。

❼ 如果是 `MutableSequence` 对象，必然是列表，[6] 因此，我们把 `obj` 中的每个元素递归地传给 `.build()` 方法，构建一个列表。

> [6]数据源是 JSON 格式，而在 JSON 中，只有字典和列表是集合类型。

❽ 如果既不是字典也不是列表，那么原封不动地返回元素。

注意，我们没有缓存或转换原始数据源。在迭代数据源的过程中，嵌套的数据结构不断被转换成 `FrozenJSON` 对象。这么做没问题，因为数据集不大，而且这个脚本只用于访问或转换数据。

从随机源中生成或仿效动态属性名的脚本都必须处理一个问题：原始数据中的键可能不适合作为属性名。下一节处理这个问题。

19.1.2　处理无效属性名

FrozenJSON 类有个缺陷：没有对名称为 Python 关键字的属性做特殊处理。比如说像下面这样构建一个对象：

```
>>> grad = FrozenJSON({'name': 'Jim Bo', 'class': 1982})
```

此时无法读取 grad.class 的值，因为在 Python 中 class 是保留字：

```
>>> grad.class
  File "<stdin>", line 1
    grad.class
             ^
SyntaxError: invalid syntax
```

当然，可以这么做：

```
>>> getattr(grad, 'class')
1982
```

但是，FrozenJSON 类的目的是为了便于访问数据，因此更好的方法是检查传给 FrozenJSON.__init__ 方法的映射中是否有键的名称为关键字，如果有，那么在键名后加上 _，然后通过下述方式读取：

```
>>> grad.class_
1982
```

为此，我们可以把示例 19-5 中只有一行代码的 __init__ 方法改成示例 19-6 中的版本。

示例 **19-6**　explore1.py：在名称为 Python 关键字的属性后面加上 _

```
def __init__(self, mapping):
    self.__data = {}
    for key, value in mapping.items():
        if keyword.iskeyword(key):  ❶
```

```
            key += '_'
        self.__data[key] = value
```

❶ keyword.iskeyword(...) 正是我们所需的函数；为了使用它，必须导入 keyword 模块；这个代码片段没有列出导入语句。

如果 JSON 对象中的键不是有效的 Python 标识符，也会遇到类似的问题：

```
>>> x = FrozenJSON({'2be':'or not'})
>>> x.2be
  File "<stdin>", line 1
    x.2be
      ^
SyntaxError: invalid syntax
```

这种有问题的键在 Python 3 中易于检测，因为 str 类提供的 s.isidentifier() 方法能根据语言的语法判断 s 是否为有效的 Python 标识符。但是，把无效的标识符变成有效的属性名却不容易。对此，有两个简单的解决方法，一个是抛出异常，另一个是把无效的键换成通用名称，例如 attr_0、attr_1，等等。为了简单起见，我将忽略这个问题。

对动态属性的名称做了一些处理之后，我们要分析 FrozenJSON 类的另一个重要功能——类方法 build 的逻辑。这个方法把嵌套结构转换成 FrozenJSON 实例或 FrozenJSON 实例列表，因此 __getattr__ 方法使用这个方法访问属性时，能为不同的值返回不同类型的对象。

除了在类方法中实现这样的逻辑之外，还可以在特殊的 __new__ 方法中实现，如下一节所述。

19.1.3　使用 __new__ 方法以灵活的方式创建对象

我们通常把 __init__ 称为构造方法，这是从其他语言借鉴过来的术语。其实，用于构建实例的是特殊方法 __new__：这是个类方法（使用特殊方式处理，因此不必使用 @classmethod 装饰器），必须返回一个实例。返回的实例会作为第一个参数（即 self）传给 __init__ 方

法。因为调用 `__init__` 方法时要传入实例，而且禁止返回任何值，所以 `__init__` 方法其实是“初始化方法”。真正的构造方法是 `__new__`。我们几乎不需要自己编写 `__new__` 方法，因为从 `object` 类继承的实现已经足够了。

刚才说明的过程，即从 `__new__` 方法到 `__init__` 方法，是最常见的，但不是唯一的。`__new__` 方法也可以返回其他类的实例，此时，解释器不会调用 `__init__` 方法。

也就是说，Python 构建对象的过程可以使用下述伪代码概括：

```
# 构建对象的伪代码
def object_maker(the_class, some_arg):
    new_object = the_class.__new__(some_arg)
    if isinstance(new_object, the_class):
        the_class.__init__(new_object, some_arg)
    return new_object

# 下述两个语句的作用基本等效
x = Foo('bar')
x = object_maker(Foo, 'bar')
```

示例 19-7 是 `FrozenJSON` 类的另一个版本，把之前在类方法 `build` 中的逻辑移到了 `__new__` 方法中。

示例 **19-7**　explore2.py：使用 `__new__` 方法取代 `build` 方法，构建可能是也可能不是 `FrozenJSON` 实例的新对象

```
from collections import abc


class FrozenJSON:
    """一个只读接口，使用属性表示法访问JSON类对象
    """

    def __new__(cls, arg):  ❶
        if isinstance(arg, abc.Mapping):
            return super().__new__(cls)  ❷
        elif isinstance(arg, abc.MutableSequence):  ❸
            return [cls(item) for item in arg]
        else:
```

```
            return arg

    def __init__(self, mapping):
        self.__data = {}
        for key, value in mapping.items():
            if iskeyword(key):
                key += '_'
            self.__data[key] = value

    def __getattr__(self, name):
        if hasattr(self.__data, name):
            return getattr(self.__data, name)
        else:
            return FrozenJSON(self.__data[name])  ❹
```

❶ `__new__` 是类方法，第一个参数是类本身，余下的参数与 `__init__` 方法一样，只不过没有 `self`。

❷ 默认的行为是委托给超类的 `__new__` 方法。这里调用的是 `object` 基类的 `__new__` 方法，把唯一的参数设为 `FrozenJSON`。

❸ `__new__` 方法中余下的代码与原先的 `build` 方法完全一样。

❹ 之前，这里调用的是 `FrozenJSON.build` 方法，现在只需调用 `FrozenJSON` 构造方法。

`__new__` 方法的第一个参数是类，因为创建的对象通常是那个类的实例。所以，在 `FrozenJSON.__new__` 方法中，`super().__new__(cls)` 表达式会调用 `object.__new__(FrozenJSON)`，而 `object` 类构建的实例其实是 `FrozenJSON` 实例，即那个实例的 `__class__` 属性存储的是 `FrozenJSON` 类的引用。不过，真正的构建操作由解释器调用 C 语言实现的 `object.__new__` 方法执行。

OSCON 的 JSON 数据源有一个明显的缺点：索引为 `40` 的事件，即名为 `'There *Will* Be Bugs'` 的那个，有两位演讲者，`3471` 和 `5199`，但却不容易找到他们，因为提供的是编号，而 `Schedule.speakers` 列表没有使用编号建立索引。此外，每条事件记录中都有 `venue_serial` 字段，存储的值也是编号，但是如果想找到对应的记录，那就要线性搜

索 Schedule.venues 列表。接下来的任务是，调整数据结构，以便自动获取所链接的记录。

19.1.4　使用shelve模块调整OSCON数据源的结构

标准库中有个 shelve（架子）模块，这名字听起来怪怪的，可是如果知道 pickle（泡菜）是 Python 对象序列化格式的名字，还是在那个格式与对象之间相互转换的某个模块的名字，就会觉得以 shelve 命名是合理的。泡菜坛子摆放在架子上，因此 shelve 模块提供了 pickle 存储方式。

shelve.open 高阶函数返回一个 shelve.Shelf 实例，这是简单的键值对象数据库，背后由 dbm 模块支持，具有下述特点。

- shelve.Shelf 是 abc.MutableMapping 的子类，因此提供了处理映射类型的重要方法。
- 此外，shelve.Shelf 类还提供了几个管理 I/O 的方法，如 sync 和 close；它也是一个上下文管理器。
- 只要把新值赋予键，就会保存键和值。
- 键必须是字符串。
- 值必须是 pickle 模块能处理的对象。

shelve（https://docs.python.org/3/library/shelve.html）、dbm（https://docs
和 pickle 模块（https://docs.python.org/3/library/pickle.html）的详细用法和注意事项参见文档。现在值得关注的是，shelve 模块为识别 OSCON 的日程数据提供了一种简单有效的方式。我们将从 JSON 文件中读取所有记录，将其存在一个 shelve.Shelf 对象中，键由记录类型和编号组成（例如，'event.33950' 或 'speaker.3471'），而值是我们即将定义的 Record 类的实例。

实例 19-8 是 schedule1.py 脚本的 doctest，使用 shelve 模块处理数据源。若想以交互式方式测试，要执行 python -i schedule1.py 命令运行脚本，启动加载了 schedule1 模块的控制台。主要工作由

load_db 函数完成：调用 osconfeed.load 方法（在示例 19-2 中定义）读取 JSON 数据，把通过 db 传入的 Shelf 对象中的各条记录存储为一个个 Record 实例。这样处理之后，获取演讲者的记录就容易了，例如 speaker = db['speaker.3471']。

示例 **19-8** 测试 schedule1.py 脚本（见示例 19-9）提供的功能

```
>>> import shelve
>>> db = shelve.open(DB_NAME)  ❶
>>> if CONFERENCE not in db:  ❷
...     load_db(db)  ❸
...
>>> speaker = db['speaker.3471']  ❹
>>> type(speaker)  ❺
<class 'schedule1.Record'>
>>> speaker.name, speaker.twitter  ❻
('Anna Martelli Ravenscroft', 'annaraven')
>>> db.close()  ❼
```

❶ shelve.open 函数打开现有的数据库文件，或者新建一个。

❷ 判断数据库是否填充的简便方法是，检查某个已知的键是否存在；这里检查的键是 conference.115，即 conference 记录（只有一个）的键。[7]

> [7]也可以使用 len(db) 判断，不过，如果是大型 dbm 数据库，那就很耗费时间。

❸ 如果数据库是空的，那就调用 load_db(db)，加载数据。

❹ 获取一条 speaker 记录。

❺ 它是示例 19-9 中定义的 Record 类的实例。

❻ 各个 Record 实例都有一系列自定义的属性，对应于底层 JSON 记录里的字段。

❼ 一定要记得关闭 shelve.Shelf 对象。如果可以，使用 with 块确保 Shelf 对象会关闭。[8]

> [8]doctest 有个突出的弱点：无法正确地设置资源并保证将其销毁。我使用 py.test 为

> schedule1.py 脚本写了很多测试，在示例 A-12 中。

schedule1.py 脚本的代码在示例 19-9 中。

示例 **19-9**　schedule1.py：访问保存在 `shelve.Shelf` 对象里的 OSCON 日程数据

```
import warnings

import osconfeed  ❶

DB_NAME = 'data/schedule1_db'
CONFERENCE = 'conference.115'


class Record:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)  ❷


def load_db(db):
    raw_data = osconfeed.load()  ❸
    warnings.warn('loading ' + DB_NAME)
    for collection, rec_list in raw_data['Schedule'].items():  ❹
        record_type = collection[:-1] ❺
        for record in rec_list:
            key = '{}.{}'.format(record_type, record['serial'])  ❻
            record['serial'] = key  ❼
            db[key] = Record(**record)  ❽
```

❶ 加载示例 19-2 中的 osconfeed.py 模块。

❷ 这是使用关键字参数传入的属性构建实例的常用简便方式（详情参见下文）。

❸ 如果本地没有副本，从网上下载 JSON 数据源。

❹ 迭代集合（例如 `'conferences'`、`'events'`，等等）。

❺ `record_type` 的值是去掉尾部 `'s'` 后的集合名（即把 `'events'` 变成 `'event'`）。

❻ 使用 `record_type` 和 `'serial'` 字段构成 `key`。

❼ 把 `'serial'` 字段的值设为完整的键。

❽ 构建 `Record` 实例，存储在数据库中的 `key` 键名下。

`Record.__init__` 方法展示了一个流行的 Python 技巧。我们知道，对象的 `__dict__` 属性中存储着对象的属性——前提是类中没有声明 `__slots__` 属性，如 9.8 节所述。因此，更新实例的 `__dict__` 属性，把值设为一个映射，能快速地在那个实例中创建一堆属性。[9]

> [9]顺便说一下，2001 年 Alex Martelli 在“The simple but handy‘collector of a bunch of named stuff’class ”诀窍（http://code.activestate.com/recipes/52308-the-simple-but-handy-collector-of-a-bunch-of-named/）中分享这个技巧时使用的类名是 `Bunch`。

我不会重述 19.1.2 节讨论的细节，不过要知道，在某些应用中，`Record` 类可能要处理不能作为属性名使用的键。

示例 19-9 中定义的 `Record` 类太简单了，因此你可能会问，为什么之前没用，而是使用更复杂的 `FrozenJSON` 类。原因有两个。第一，`FrozenJSON` 类要递归转换嵌套的映射和列表；而 `Record` 类不需要这么做，因为转换好的数据集中没有嵌套的映射和列表，记录中只有字符串、整数、字符串列表和整数列表。第二，`FrozenJSON` 类要访问内嵌的 `__data` 属性（值是字典，用于调用 `keys` 等方法），而现在我们也不需要这么做了。

Python 标准库中至少有两个与 `Record` 类似的类，其实例可以有任意个属性，由传给构造方法的关键字参数构建——`multiprocessing.Namespace` 类 [文档（https://docs.python.org/3/library/multiprocessing.html?highlight=namespace#namespaceobjects），源码（https://hg.python.org/cpython/file/50d581f69a73/Lib/multiprocessing/man
和 `argparse.Namespace` 类 [文档（https://docs.python.org/3/library/argparse.html#argparse.Namespace），源码(https://hg.python.org/cpython/file/50d581f69a73/Lib/argparse.py#l1196）]

我之所以自己实现 `Record`，是为了说明一个重要的做法：在 `__init__` 方法中更新实例的 `__dict__` 属性。

像上面那样调整日程数据集之后，我们可以扩展 `Record` 类，让它提供一个有用的服务：自动获取 `event` 记录引用的 `venue` 和 `speaker` 记录。这与 Django ORM 访问 `models.ForeignKey` 字段时所做的事类似：得到的不是键，而是链接的模型对象。在下一个示例中，我们要使用特性来实现这个服务。

19.1.5　使用特性获取链接的记录

下一个版本的目标是，对于从 `Shelf` 对象中获取的 `event` 记录来说，读取它的 `venue` 或 `speakers` 属性时返回的不是编号，而是完整的记录对象。用法如示例 19-10 中的交互代码片段所示。

示例 **19-10**　摘自 schedule2.py 脚本的 doctest

```
>>> DbRecord.set_db(db)  ❶
>>> event = DbRecord.fetch('event.33950')  ❷
>>> event  ❸
<Event 'There *Will* Be Bugs'>
>>> event.venue  ❹
<DbRecord serial='venue.1449'>
>>> event.venue.name  ❺
'Portland 251'
>>> for spkr in event.speakers:  ❻
...     print('{0.serial}: {0.name}'.format(spkr))
...
speaker.3471: Anna Martelli Ravenscroft
speaker.5199: Alex Martelli
```

❶ `DbRecord` 类扩展 `Record` 类，添加对数据库的支持：为了操作数据库，必须为 `DbRecord` 提供一个数据库的引用。

❷ `DbRecord.fetch` 类方法能获取任何类型的记录。

❸ 注意，`event` 是 `Event` 类的实例，而 `Event` 类扩展 `DbRecord` 类。

❹ `event.venue` 返回一个 `DbRecord` 实例。

❺ 现在，想找出 `event.venue` 的名称就容易了。这种自动取值是这个示例的目标。

❻ 还可以迭代 `event.speakers` 列表，获取表示各位演讲者的 `DbRecord` 对象。

图 19-1 绘出了本节要分析的几个类。

`Record`

　　`__init__` 方法与 schedule1.py 脚本（见示例 19-9）中的一样；为了辅助测试，增加了 `__eq__` 方法。

`DbRecord`

　　`Record` 类的子类，添加了 `__db` 类属性，用于设置和获取 `__db` 属性的 `set_db` 和 `get_db` 静态方法，用于从数据库中获取记录的 `fetch` 类方法，以及辅助调试和测试的 `__repr__` 实例方法。

`Event`

　　`DbRecord` 类的子类，添加了用于获取所链接记录的 `venue` 和 `speakers` 属性，以及特殊的 `__repr__` 方法。

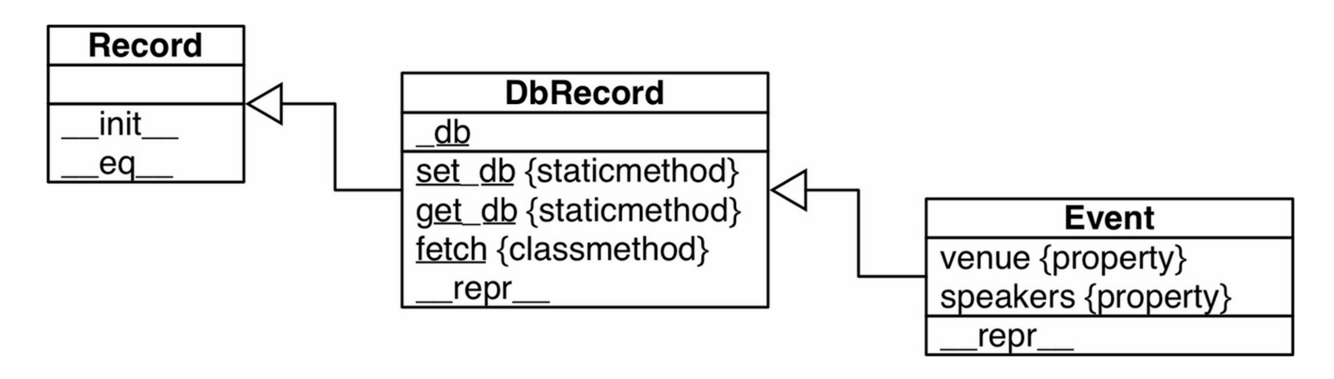


图 19-1：改进的 `Record` 类和两个子类（`DbRecord` 和 `Event`）的 UML 类图

`DbRecord.__db` 类属性的作用是存储打开的 `shelve.Shelf` 数据库引用，以便在需要使用数据库的 `DbRecord.fetch` 方法及 `Event.venue` 和 `Event.speakers` 属性中使用。我把 `__db` 设为私有类属性，然后定

义了普通的读值方法和设值方法，以防不小心覆盖 __db 属性的值。基于一个重要的原因，我没有使用特性去管理 __db 属性：特性是用于管理实例属性的类属性。[10]

> [10]Stack Overflow 中有个题为“Class-level read only properties in Python”的问题（http://stackoverflow.com/questions/1735434/class-level-read-only-properties-in-python），为类中的只读属性提供了解决方案，其中包括 Alex Martelli 提供的一个方案。这些方案要用到元类，因此学习那些方案之前可能要先读本书第 21 章。

本节的代码在本书仓库（https://github.com/fluentpython/example-code）里的 schedule2.py 模块中。这个模块有 100 多行，因此我会分成几部分分析。

schedule2.py 脚本的前几个语句在示例 19-11 中。

示例 **19-11** schedule2.py：导入模块，定义常量和增强的 Record 类

```
import warnings
import inspect  ❶

import osconfeed

DB_NAME = 'data/schedule2_db'  ❷
CONFERENCE = 'conference.115'


class Record:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

    def __eq__(self, other):  ❸
        if isinstance(other, Record):
            return self.__dict__ == other.__dict__
        else:
            return NotImplemented
```

❶ inspect 模块在 load_db 函数中使用（参见示例 19-14）。

❷ 因为要存储几个不同类的实例，所以我们要创建并使用不同的数据库文件；这里不用示例 19-9 中的 'schedule1_db'，而是使用

'schedule2_db'。

❸ __eq__ 方法对测试有重大帮助。



在 Python 2 中，只有“新式”类支持特性。在 Python 2 中定义新式类的方法是，直接或间接继承 object 类。示例 19-11 中的 Record 类是一个继承体系的基类，用到了特性；因此，在 Python 2 中声明 Record 类时，开头要这么写：[11]

```
class Record(object):
    # 余下的代码……
```

[11]在 Python 3 中明确指明继承 object 类没有错，但是多余，因为现在所有类都是新式的。此例说明，与过去告别能让语言更简洁。如果要在 Python 2 和 Python 3 中运行同一段代码，应该显式继承 object 类。

接下来，schedule2.py 脚本定义了两个类——一个自定义的异常类型和 DbRecord 类，参见示例 19-12。

示例 **19-12**　schedule2.py：MissingDatabaseError 类和 DbRecord 类

```
class MissingDatabaseError(RuntimeError):
    """需要数据库但没有指定数据库时抛出。"""  ❶


class DbRecord(Record):  ❷

    __db = None  ❸

    @staticmethod  ❹
    def set_db(db):
        DbRecord.__db = db  ❺

    @staticmethod  ❻
    def get_db():
        return DbRecord.__db
```

```
    @classmethod  ❼
    def fetch(cls, ident):
        db = cls.get_db()
        try:
            return db[ident]  ❽
        except TypeError:
            if db is None:  ❾
                msg = "database not set; call '{}.set_db(my_db)'"
                raise MissingDatabaseError(msg.format(cls.__name__))
            else:  ❿
                raise

    def __repr__(self):
        if hasattr(self, 'serial'):  ⓫
            cls_name = self.__class__.__name__
            return '<{} serial={!r}>'.format(cls_name, self.serial)
        else:
            return super().__repr__()  ⓬
```

❶ 自定义的异常通常是标志类，没有定义体。写一个文档字符串，说明异常的用途，比只写一个 pass 语句要好。

❷ DbRecord 类扩展 Record 类。

❸ __db 类属性存储一个打开的 shelve.Shelf 数据库引用。

❹ set_db 是静态方法，以此强调不管调用多少次，效果始终一样。

❺ 即使调用 Event.set_db(my_db)，__db 属性仍在 DbRecord 类中设置。

❻ get_db 也是静态方法，因为不管怎样调用，返回值始终是 DbRecord.__db 引用的对象。

❼ fetch 是类方法，因此在子类中易于定制它的行为。

❽ 从数据库中获取 ident 键对应的记录。

❾ 如果捕获到 TypeError 异常，而且 db 变量的值是 None，抛出自定义的异常，说明必须设置数据库。

❿ 否则，重新抛出 TypeError 异常，因为我们不知道怎么处理。

⓫ 如果记录有 serial 属性，在字符串表示形式中使用。

⓬ 否则，调用继承的 __repr__ 方法。

现在到这个示例的重要部分了——Event 类，如示例 19-13 所示。

示例 **19-13**　schedule2.py：Event 类

```
class Event(DbRecord):  ❶

    @property
    def venue(self):
        key = 'venue.{}'.format(self.venue_serial)
        return self.__class__.fetch(key)  ❷

    @property
    def speakers(self):
        if not hasattr(self, '_speaker_objs'):  ❸
            spkr_serials = self.__dict__['speakers']  ❹
            fetch = self.__class__.fetch  ❺
            self._speaker_objs = [fetch('speaker.{}'.format(key))
                                  for key in spkr_serials]  ❻
        return self._speaker_objs  ❼

    def __repr__(self):
        if hasattr(self, 'name'):  ❽
            cls_name = self.__class__.__name__
            return '<{} {!r}>'.format(cls_name, self.name)
        else:
            return super().__repr__()  ❾
```

❶ Event 类扩展 DbRecord 类。

❷ 在 venue 特性中使用 venue_serial 属性构建 key，然后传给继承自 DbRecord 类的 fetch 类方法（详情参见下文）。

❸ speakers 特性检查记录是否有 _speaker_objs 属性。

❹ 如果没有，直接从 __dict__ 实例属性中获取 'speakers' 属性的

值，防止无限递归，因为这个特性的公开名称也是 `speakers`。

❺ 获取 `fetch` 类方法的引用（稍后会说明这么做的原因）。

❻ 使用 `fetch` 获取 `speaker` 记录列表，然后赋值给 `self._speaker_objs`。

❼ 返回前面获取的列表。

❽ 如果记录有 `name` 属性，在字符串表示形式中使用。

❾ 否则，调用继承的 `__repr__` 方法。

在示例 19-13 中的 `venue` 特性里，最后一行返回的是 `self.__class__.fetch(key)`，为什么不直接使用 `self.fetch(key)` 呢？对这个 OSCON 数据源来说，可以使用后者，因为事件记录都没有 `'fetch'` 键。哪怕只有一个事件记录有名为 `'fetch'` 的键，那么在那个 `Event` 实例中，`self.fetch` 获取的是 `fetch` 字段的值，而不是 `Event` 继承自 `DbRecord` 的 `fetch` 类方法。这个缺陷不明显，很容易被测试忽略；在生产环境中，如果会场或演讲者记录链接到那个事件记录，获取事件记录时才会暴露出来。

> 从数据中创建实例属性的名称时肯定有可能会引入缺陷，因为类属性（例如方法）可能被遮盖，或者由于意外覆盖现有的实例属性而丢失数据。这个问题可能是 Python 字典默认不能像 JavaScript 对象那样访问的主要原因。

如果 `Record` 类的行为更像映射，可以把动态的 `__getattr__` 方法换成动态的 `__getitem__` 方法，这样就不会出现由于覆盖或遮盖而引起的缺陷了。使用映射实现 `Record` 类或许更符合 Python 风格。可是，如果我采用那种方式，就发掘不了动态属性编程的技巧和陷阱了。

这个示例最后的代码是重写的 `load_db` 函数，如示例 19-14。

示例 **19-14**　schedule2.py：`load_db` 函数

```
def load_db(db):
```

```
    raw_data = osconfeed.load()
    warnings.warn('loading ' + DB_NAME)
    for collection, rec_list in raw_data['Schedule'].items():
        record_type = collection[:-1]  ❶
        cls_name = record_type.capitalize()  ❷
        cls = globals().get(cls_name, DbRecord)  ❸
        if inspect.isclass(cls) and issubclass(cls, DbRecord):  ❹
            factory = cls  ❺
        else:
            factory = DbRecord  ❻
        for record in rec_list:  ❼
            key = '{}.{}'.format(record_type, record['serial'])
            record['serial'] = key
            db[key] = factory(**record)  ❽
```

❶ 目前，与 schedule1.py 脚本（见示例 19-9）中的 `load_db` 函数一样。

❷ 把 `record_type` 变量的值首字母变成大写（例如，把 `'event'` 变成 `'Event'`），获取可能的类名。

❸ 从模块的全局作用域中获取那个名称对应的对象；如果找不到对象，使用 `DbRecord`。

❹ 如果获取的对象是类，而且是 `DbRecord` 的子类……

❺ ……把对象赋值给 `factory` 变量。因此，`factory` 的值可能是 `DbRecord` 的任何一个子类，具体的类取决于 `record_type` 的值。

❻ 否则，把 `DbRecord` 赋值给 `factory` 变量。

❼ 这个 `for` 循环创建 `key`，然后保存记录，这与之前一样，不过……

❽ ……存储在数据库中的对象由 `factory` 构建，`factory` 可能是 `DbRecord` 类，也可能是根据 `record_type` 的值确定的某个子类。

注意，只有事件类型的记录有自定义的类——`Event`。不过，如果定义了 `Speaker` 或 `Venue` 类，`load_db` 函数构建和保存记录时会自动使用这两个类，而不会使用默认的 `DbRecord` 类。

本章目前所举的示例是为了展示如何使用基本的工具，如 `__getattr__` 方法、`hasattr` 函数、`getattr` 函数、`@property` 装饰器和 `__dict__` 属性，来实现动态属性。

特性经常用于把公开的属性变成使用读值方法和设值方法管理的属性，且在不影响客户端代码的前提下实施业务规则，如下一节所述。

19.2　使用特性验证属性

目前，我们只介绍了如何使用 `@property` 装饰器实现只读特性。本节要创建一个可读写的特性。

19.2.1　`LineItem`类第1版：表示订单中商品的类

假设有个销售散装有机食物的电商应用，客户可以按重量订购坚果、干果或杂粮。在这个系统中，每个订单中都有一系列商品，而每个商品都可以使用示例 19-15 中的类表示。

示例 **19-15**　bulkfood_v1：最简单的 `LineItem` 类

```
class LineItem:

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

这个类很精简，不过或许太简单了。示例 19-16 揭示了一个问题。

示例 **19-16**　重量为负值时，金额小计为负值

```
    >>> raisins = LineItem('Golden raisins', 10, 6.95)
    >>> raisins.subtotal()
    69.5
    >>> raisins.weight = -20  # 无效输入……
    >>> raisins.subtotal()    # 无效输出……
    -139.0
```

这个示例像玩具一样，但是没有想象中的那么好玩。下面是亚马逊早期的真实故事。

> 我们发现顾客买书时可以把数量设为负数！然后，我们把金额打到顾客的信用卡上，苦苦等待他们把书寄出（想得美）。[12]
>
> ——Jeff Bezos
> 亚马逊创始人和 CEO

[12]摘自《华尔街日报》的文章，“Birth of a Salesman”（2011 年 10 月 15 日，http://www.wsj.com/articles/SB10001424052970203914304576627102996831200），这是 Jeff Bezos 的原话。

这个问题怎么解决呢？我们可以修改 `LineItem` 类的接口，使用读值方法和设值方法管理 `weight` 属性。这是 Java 采用的方式，这里也完全可行。

但是，如果能直接设定商品的 `weight` 属性，显得更自然。此外，系统可能在生产环境中，而其他部分已经直接访问 `item.weight` 了。此时，符合 Python 风格的做法是，把数据属性换成特性。

19.2.2　LineItem类第2版：能验证值的特性

实现特性之后，我们可以使用读值方法和设值方法，但是 `LineItem` 类的接口保持不变（即，设置 `LineItem` 对象的 `weight` 属性依然写成 `raisins.weight = 12`）。

示例 19-17 列出可读写的 `weight` 特性的代码。

示例 **19-17**　bulkfood_v2.py：定义了 `weight` 特性的 `LineItem` 类

```
class LineItem:

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight  ❶
        self.price = price

    def subtotal(self):
        return self.weight * self.price

    @property  ❷
    def weight(self):  ❸
```

```
        return self.__weight  ❹

    @weight.setter  ❺
    def weight(self, value):
        if value > 0:
            self.__weight = value  ❻
        else:
            raise ValueError('value must be > 0')  ❼
```

❶ 这里已经使用特性的设值方法了，确保所创建实例的 `weight` 属性不能为负值。

❷ `@property` 装饰读值方法。

❸ 实现特性的方法，其名称都与公开属性的名称一样——`weight`。

❹ 真正的值存储在私有属性 `__weight` 中。

❺ 被装饰的读值方法有个 `.setter` 属性，这个属性也是装饰器；这个装饰器把读值方法和设值方法绑定在一起。

❻ 如果值大于零，设置私有属性 `__weight`。

❼ 否则，抛出 `ValueError` 异常。

注意，现在不能创建重量为无效值的 `LineItem` 对象：

```
>>> walnuts = LineItem('walnuts', 0, 10.00)
Traceback (most recent call last):
    ...
ValueError: value must be > 0
```

现在，我们禁止用户为 `weight` 属性提供负值或零。虽然买家通常不能设置商品的价格，但是工作人员可能犯错，应用程序也可能有缺陷，从而导致 `LineItem` 对象的 `price` 属性为负值。为了防止出现这种情况，我们也可以把 `price` 属性变成特性，但是这样我们的代码中就存在一些重复。

还记得第 14 章引述 Paul Graham 的那句话吗？他说：“当我在自己的程

序中发现用到了模式，我觉得这就表明某个地方出错了。”去除重复的方法是抽象。抽象特性的定义有两种方式：使用特性工厂函数，或者使用描述符类。后者更灵活，第 20 章会全面讨论。其实，特性本身就是使用描述符类实现的。不过，这里我们要继续探讨特性，实现一个特性工厂函数。

但是，在实现特性工厂函数之前，我们要深入理解特性。

19.3　特性全解析

虽然内置的 property 经常用作装饰器，但它其实是一个类。在 Python 中，函数和类通常可以互换，因为二者都是可调用的对象，而且没有实例化对象的 new 运算符，所以调用构造方法与调用工厂函数没有区别。此外，只要能返回新的可调用对象，代替被装饰的函数，二者都可以用作装饰器。

property 构造方法的完整签名如下：

```
property(fget=None, fset=None, fdel=None, doc=None)
```

所有参数都是可选的，如果没有把函数传给某个参数，那么得到的特性对象就不允许执行相应的操作。

property 类型在 Python 2.2 中引入，但是直到 Python 2.4 才出现 @ 装饰器句法，因此有那么几年，若想定义特性，则只能把存取函数传给前两个参数。

不使用装饰器定义特性的“经典”句法如示例 19-18 所示。

示例 **19-18**　bulkfood_v2b.py：效果与示例 19-17 一样，只不过没使用装饰器

```
class LineItem:

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price

    def get_weight(self):  ❶
        return self.__weight
```

```
    def set_weight(self, value):  ❷
        if value > 0:
            self.__weight = value
        else:
            raise ValueError('value must be > 0')

    weight = property(get_weight, set_weight)  ❸
```

❶ 普通的读值方法。

❷ 普通的设值方法。

❸ 构建 property 对象，然后赋值给公开的类属性。

某些情况下，这种经典形式比装饰器句法好；稍后讨论的特性工厂函数就是一例。但是，在方法众多的类定义体中使用装饰器的话，一眼就能看出哪些是读值方法，哪些是设值方法，而不用按照惯例，在方法名的前面加上 get 和 set。

类中的特性能影响实例属性的寻找方式，而一开始这种方式可能会让人觉得意外。下一节会详细说明。

19.3.1　特性会覆盖实例属性

特性都是类属性，但是特性管理的其实是实例属性的存取。

9.9 节说过，如果实例和所属的类有同名数据属性，那么实例属性会覆盖（或称遮盖）类属性——至少通过那个实例读取属性时是这样。示例 19-19 阐明了这一点。

示例 **19-19**　实例属性遮盖类的数据属性

```
>>> class Class:   # ❶
...     data = 'the class data attr'
...     @property
...     def prop(self):
...         return 'the prop value'
...
>>> obj = Class()
>>> vars(obj)  # ❷
```

```
{}
>>> obj.data  # ❸
'the class data attr'
>>> obj.data = 'bar' # ❹
>>> vars(obj)  # ❺
{'data': 'bar'}
>>> obj.data  # ❻
'bar'
>>> Class.data  # ❼
'the class data attr'
```

❶ 定义 Class 类，这个类有两个类属性：data 数据属性和 prop 特性。

❷ vars 函数返回 obj 的 __dict__ 属性，表明没有实例属性。

❸ 读取 obj.data，获取的是 Class.data 的值。

❹ 为 obj.data 赋值，创建一个实例属性。

❺ 审查实例，查看实例属性。

❻ 现在读取 obj.data，获取的是实例属性的值。从 obj 实例中读取属性时，实例属性 data 会遮盖类属性 data。

❼ Class.data 属性的值完好无损。

下面尝试覆盖 obj 实例的 prop 特性。接着前面的控制台会话，输入示例 19-20 中的代码。

示例 **19-20**　实例属性不会遮盖类特性（接续示例 19-19）

```
>>> Class.prop  # ❶
<property object at 0x1072b7408>
>>> obj.prop  # ❷
'the prop value'
>>> obj.prop = 'foo'  # ❸
Traceback (most recent call last):
  ...
AttributeError: can't set attribute
>>> obj.__dict__['prop'] = 'foo'  # ❹
>>> vars(obj)  # ❺
```

```
{ 'data': 'bar','prop': 'foo'}
>>> obj.prop  # ❻
'the prop value'
>>> Class.prop = 'baz'  # ❼
>>> obj.prop  # ❽
'foo'
```

❶ 直接从 `Class` 中读取 `prop` 特性，获取的是特性对象本身，不会运行特性的读值方法。

❷ 读取 `obj.prop` 会执行特性的读值方法。

❸ 尝试设置 `prop` 实例属性，结果失败。

❹ 但是可以直接把 `'prop'` 存入 `obj.__dict__`。

❺ 可以看到，`obj` 现在有两个实例属性：`data` 和 `prop`。

❻ 然而，读取 `obj.prop` 时仍会运行特性的读值方法。特性没被实例属性遮盖。

❼ 覆盖 `Class.prop` 特性，销毁特性对象。

❽ 现在，`obj.prop` 获取的是实例属性。`Class.prop` 不是特性了，因此不会再覆盖 `obj.prop`。

最后再举一个例子，为 `Class` 类新添一个特性，覆盖实例属性。示例 19-21 接续示例 19-20。

示例 **19-21**　新添的类特性遮盖现有的实例属性（接续示例 19-20）

```
>>> obj.data  # ❶
'bar'
>>> Class.data  # ❷
'the class data attr'
>>> Class.data = property(lambda self: 'the "data" prop value')  # ❸
>>> obj.data  # ❹
'the "data" prop value'
>>> del Class.data  # ❺
>>> obj.data  # ❻
```

```
'bar'
```

❶ obj.data 获取的是实例属性 data。

❷ Class.data 获取的是类属性 data。

❸ 使用新特性覆盖 Class.data。

❹ 现在，obj.data 被 Class.data 特性遮盖了。

❺ 删除特性。

❻ 现在恢复原样，obj.data 获取的是实例属性 data。

本节的主要观点是，obj.attr 这样的表达式不会从 obj 开始寻找 attr，而是从 obj.__class__ 开始，而且，仅当类中没有名为 attr 的特性时，Python 才会在 obj 实例中寻找。这条规则不仅适用于特性，还适用于一整类描述符——覆盖型描述符（overriding descriptor）。第 20 章会进一步讨论描述符，那时你会发现，特性其实是覆盖型描述符。

现在回到特性。各种 Python 代码单元（模块、函数、类和方法）都可以有文档字符串。下一节说明如何把文档依附到特性上。

19.3.2　特性的文档

控制台中的 help() 函数或 IDE 等工具需要显示特性的文档时，会从特性的 __doc__ 属性中提取信息。

如果使用经典调用句法，为 property 对象设置文档字符串的方法是传入 doc 参数：

```
weight = property(get_weight, set_weight, doc='weight in kilograms')
```

使用装饰器创建 property 对象时，读值方法（有 @property 装饰器的方法）的文档字符串作为一个整体，变成特性的文档。图 19-2 显示

的是从示例 19-22 里的代码中生成的帮助界面。

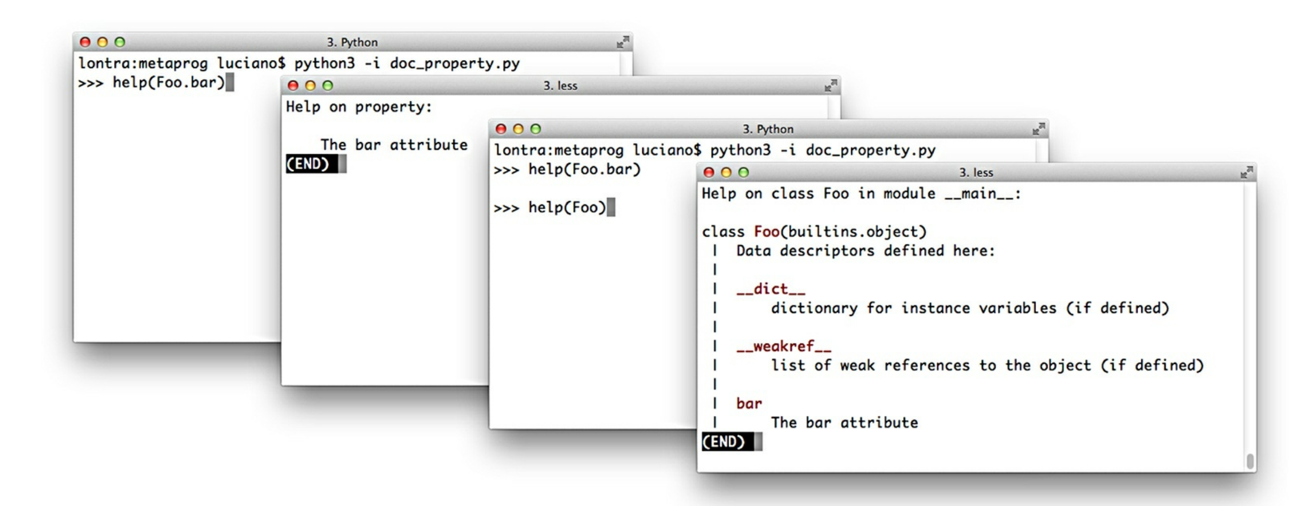


图 **19-2**：在 **Python** 控制台中执行 **`help(Foo.bar)`** 和 **`help(Foo)`** 命令时的截图；源码在示例 **19-22** 中

示例 **19-22**　特性的文档

```
class Foo:

    @property
    def bar(self):
        '''The bar attribute'''
        return self.__dict__['bar']

    @bar.setter
    def bar(self, value):
        self.__dict__['bar'] = value
```

至此，我们介绍了特性的重要知识。下面回过头来解决前面遇到的问题：保护 `LineItem` 对象的 `weight` 和 `price` 属性，只允许设为大于零的值；但是，不用手动实现两对几乎一样的读值方法和设值方法。

19.4 定义一个特性工厂函数

我们将定义一个名为 `quantity` 的特性工厂函数，取这个名字是因为，在这个应用中要管理的属性表示不能为负数或零的量。示例 19-23 是 `LineItem` 类的简洁版，用到了 `quantity` 特性的两个实例：一个用于管理 `weight` 属性，另一个用于管理 `price` 属性。

示例 **19-23** bulkfood_v2prop.py：使用特性工厂函数 `quantity`

```
class LineItem:
    weight = quantity('weight')  ❶
    price = quantity('price')  ❷

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight  ❸
        self.price = price

    def subtotal(self):
        return self.weight * self.price  ❹
```

❶ 使用工厂函数把第一个自定义的特性 `weight` 定义为类属性。

❷ 第二次调用，构建另一个自定义的特性，`price`。

❸ 这里，特性已经激活，确保不能把 `weight` 设为负数或零。

❹ 这里也用到了特性，使用特性获取实例中存储的值。

前文说过，特性是类属性。构建各个 `quantity` 特性对象时，要传入 `LineItem` 实例属性的名称，让特性管理。可惜，这一行要两次输入单词 `weight`：

```
    weight = quantity('weight')
```

这里很难避免重复输入，因为特性根本不知道要绑定哪个类属性名。记

住，赋值语句的右边先计算，因此调用 quantity() 时，weight 类属性还不存在。

如果想改进 quantity 特性，避免用户重复输入属性名，那么对元编程来说是个挑战。第 20 章会介绍一种变通方法，真正的解决方法在第 21 章说明，因为要么得使用类装饰器，要么得使用元类。

示例 19-24 列出 quantity 特性工厂函数的实现。[13]

[13]这段代码改编自 David Beazley 与 Brian K. Jones 的《Python Cookbook（第 3 版）中文版》一书中的“9.21 避免出现重复的属性方法”一节。

示例 **19-24**　bulkfood_v2prop.py：quantity 特性工厂函数

```
def quantity(storage_name):  ❶

    def qty_getter(instance):  ❷
        return instance.__dict__[storage_name]  ❸

    def qty_setter(instance, value):  ❹
        if value > 0:
            instance.__dict__[storage_name] = value  ❺
        else:
            raise ValueError('value must be > 0')

    return property(qty_getter, qty_setter)  ❻
```

❶ storage_name 参数确定各个特性的数据存储在哪儿；对 weight 特性来说，存储的名称是 'weight'。

❷ qty_getter 函数的第一个参数可以命名为 self，但是这么做很奇怪，因为 qty_getter 函数不在类定义体中；instance 指代要把属性存储其中的 LineItem 实例。

❸ qty_getter 引用了 storage_name，把它保存在这个函数的闭包里；值直接从 instance.__dict__ 中获取，为的是跳过特性，防止无限递归。

❹ 定义 qty_setter 函数，第一个参数也是 instance。

❺ 值直接存到 instance.__dict__ 中，这也是为了跳过特性。

❻ 构建一个自定义的特性对象，然后将其返回。

示例 19-24 中值得仔细分析的代码是与 storage_name 变量相关的部分。使用传统方式定义特性时，用于存储值的属性名硬编码在读值方法和设值方法中。但是，这里的 qty_getter 和 qty_setter 函数是通用的，要依靠 storage_name 变量判断从 __dict__ 中获取哪个属性，或者设置哪个属性。每次调用 quantity 工厂函数构建属性时，都要把 storage_name 参数设为独一无二的值。

在工厂函数的最后一行，我们使用 property 对象包装 qty_getter 和 qty_setter 函数。需要运行这两个函数时，它们会从闭包中读取 storage_name，确定从哪里获取属性的值，或者在哪里存储属性的值。

在示例 19-25 中，我创建并审查了一个 LineItem 示例，说明存储值的是哪个属性。

示例 **19-25**　bulkfood_v2prop.py：quantity 特性工厂函数

```
>>> nutmeg = LineItem('Moluccan nutmeg', 8, 13.95)
>>> nutmeg.weight, nutmeg.price  ❶
(8, 13.95)
>>> sorted(vars(nutmeg).items())  ❷
[('description', 'Moluccan nutmeg'), ('price', 13.95), ('weight', 8)]
```

❶ 通过特性读取 weight 和 price，这会遮盖同名实例属性。

❷ 使用 vars 函数审查 nutmeg 实例，查看真正用于存储值的实例属性。

注意，工厂函数构建的特性利用了 19.3.1 节所述的行为：weight 特性覆盖了 weight 实例属性，因此对 self.weight 或 nutmeg.weight 的每个引用都由特性函数处理，只有直接存取 __dict__ 属性才能跳过特性的处理逻辑。

示例 19-25 中的代码有点难理解，不过够简洁，与示例 19-17 中使用装饰器声明读值方法和设值方法的代码行数一样，但是那里只定义了 `weight` 特性。示例 19-23 中定义的 `LineItem` 类没有干扰人的读值方法和设值方法，看起来舒服多了。

在真实的系统中，分散在多个类中的多个字段可能要做同样的验证，此时最好把 `quantity` 工厂函数放在实用工具模块中，以便重复使用。最终可能要重构那个简单的工厂函数，改成更易扩展的描述符类，然后使用专门的子类执行不同的验证。在第 20 章中，我们会这么做。

下面要分析删除属性的问题，以此结束对特性的讨论。

19.5　处理属性删除操作

学过 Python 教程，我们知道，对象的属性可以使用 `del` 语句删除：

```
del my_object.an_attribute
```

其实，使用 Python 编程时不常删除属性，通过特性删除属性更少见。但是，Python 支持这么做，我可以虚构一个示例，演示这种处理方式。

定义特性时，可以使用 `@my_propety.deleter` 装饰器包装一个方法，负责删除特性管理的属性。下面兑现承诺，虚构一个示例，说明如何定义特性删值方法，如示例 19-26 所示。

示例 **19-26**　blackknight.py：灵感来自电影《巨蟒与圣杯》中的黑衣骑士角色

```
class BlackKnight:

    def __init__(self):
        self.members = ['an arm', 'another arm',
                        'a leg', 'another leg']
        self.phrases = ["'Tis but a scratch.",
                        "It's just a flesh wound.",
                        "I'm invincible!",
                        "All right, we'll call it a draw."]

    @property
    def member(self):
        print('next member is:')
        return self.members[0]

    @member.deleter
    def member(self):
        text = 'BLACK KNIGHT (loses {})\n-- {}'
        print(text.format(self.members.pop(0), self.phrases.pop(0)))
```

blackknight.py 脚本的 doctest 在示例 19-27 中。

示例 **19-27**　blackknight.py：示例 19-26 的 doctest（黑衣骑士从不屈服）

```
>>> knight = BlackKnight()
>>> knight.member
next member is:
'an arm'
>>> del knight.member
BLACK KNIGHT (loses an arm)
-- 'Tis but a scratch.
>>> del knight.member
BLACK KNIGHT (loses another arm)
-- It's just a flesh wound.
>>> del knight.member
BLACK KNIGHT (loses a leg)
-- I'm invincible!
>>> del knight.member
BLACK KNIGHT (loses another leg)
-- All right, we'll call it a draw.
```

在不使用装饰器的经典调用句法中，`fdel` 参数用于设置删值函数。例如，在 `BlackKnight` 类的定义体中可以像下面这样创建 `member` 特性：

```
member = property(member_getter, fdel=member_deleter)
```

如果不使用特性，还可以实现低层特殊的 `__delattr__` 方法处理删除属性的操作，参见 19.6.3 节。留给喜欢拖延的读者一个练习：虚构一个类，定义 `__delattr__` 方法。

特性是个强大的功能，不过有时更适合使用简单的或底层的替代方案。在本章的最后一节中，我们将回顾 Python 为动态属性编程提供的部分核心 API。

19.6　处理属性的重要属性和函数

本章及本书前面的章节多次用到 Python 为处理动态属性而提供的内置函数和特殊的方法。这些函数和方法的文档散布在官方文档中，因此我专门写了一节集中介绍它们。

19.6.1　影响属性处理方式的特殊属性

后面几节中的很多函数和特殊方法，其行为受下述 3 个特殊属性的影响。

`__class__`

　　对象所属类的引用（即 `obj.__class__` 与 `type(obj)` 的作用相同）。Python 的某些特殊方法，例如 `__getattr__`，只在对象的类中寻找，而不在实例中寻找。

`__dict__`

　　一个映射，存储对象或类的可写属性。有 `__dict__` 属性的对象，任何时候都能随意设置新属性。如果类有 `__slots__` 属性，它的实例可能没有 `__dict__` 属性。参见下面对 `__slots__` 属性的说明。

`__slots__`

　　类可以定义这个这属性，限制实例能有哪些属性。`__slots__` 属性的值是一个字符串组成的元组，指明允许有的属性。[14] 如果 `__slots__` 中没有 `'__dict__'`，那么该类的实例没有 `__dict__` 属性，实例只允许有指定名称的属性。

> [14]Alex Martelli 指出，`__slots__` 属性的值虽然可以是一个列表，但是最好始终使用元组，因为处理完类的定义体之后再修改 `__slots__` 列表没有任何作用，所以使用可变的序列容易让人误解。

19.6.2　处理属性的内置函数

下述 5 个内置函数对对象的属性做读、写和内省操作。

dir([object])

　　列出对象的大多数属性。官方文档（https://docs.python.org/3/library/functions.html#dir）说，dir 函数的目的是交互式使用，因此没有提供完整的属性列表，只列出一组“重要的”属性名。dir 函数能审查有或没有 __dict__ 属性的对象。dir 函数不会列出 __dict__ 属性本身，但会列出其中的键。dir 函数也不会列出类的几个特殊属性，例如 __mro__、__bases__ 和 __name__。如果没有指定可选的 object 参数，dir 函数会列出当前作用域中的名称。

getattr(object, name[, default])

　　从 object 对象中获取 name 字符串对应的属性。获取的属性可能来自对象所属的类或超类。如果没有指定的属性，getattr 函数抛出 AttributeError 异常，或者返回 default 参数的值（如果设定了这个参数的话）。

hasattr(object, name)

　　如果 object 对象中存在指定的属性，或者能以某种方式（例如继承）通过 object 对象获取指定的属性，返回 True。文档（https://docs.python.org/3/library/functions.html#hasattr）说道：“这个函数的实现方法是调用 getattr(object, name) 函数，看看是否抛出 AttributeError 异常。”

setattr(object, name, value)

　　把 object 对象指定属性的值设为 value，前提是 object 对象能接受那个值。这个函数可能会创建一个新属性，或者覆盖现有的属性。

vars([object])

　　返回 object 对象的 __dict__ 属性；如果实例所属的类定义了 __slots__ 属性，实例没有 __dict__ 属性，那么 vars 函数不能处理那个实例（相反，dir 函数能处理这样的实例）。如果没有指定参数，那么 vars() 函数的作用与 locals() 函数一样：返回表示本地作用域

的字典。

19.6.3　处理属性的特殊方法

在用户自己定义的类中，下述特殊方法用于获取、设置、删除和列出属性。

使用点号或内置的 `getattr`、`hasattr` 和 `setattr` 函数存取属性都会触发下述列表中相应的特殊方法。但是，直接通过实例的 `__dict__` 属性读写属性不会触发这些特殊方法——如果需要，通常会使用这种方式跳过特殊方法。

Python 文档“Data model”一章中的“3.3.9. Special method lookup”一节（https://docs.python.org/3/reference/datamodel.html#special-method-lookup）警告说：

> 对用户自己定义的类来说，如果隐式调用特殊方法，仅当特殊方法在对象所属的类型上定义，而不是在对象的实例字典中定义时，才能确保调用成功。

也就是说，要假定特殊方法从类上获取，即便操作目标是实例也是如此。因此，特殊方法不会被同名实例属性遮盖。

在下述示例中，假设有个名为 `Class` 的类，`obj` 是 `Class` 类的实例，`attr` 是 `obj` 的属性。

不管是使用点号存取属性，还是使用 19.6.2 节列出的某个内置函数，都会触发下述特殊方法中的一个。例如，`obj.attr` 和 `getattr(obj, 'attr', 42)` 都会触发 `Class.__getattribute__(obj, 'attr')` 方法。

`__delattr__(self, name)`

　　只要使用 `del` 语句删除属性，就会调用这个方法。例如，`del obj.attr` 语句触发 `Class.__delattr__(obj, 'attr')` 方法。

`__dir__(self)`

把对象传给 `dir` 函数时调用，列出属性。例如，`dir(obj)` 触发 `Class.__dir__(obj)` 方法。

`__getattr__(self, name)`

仅当获取指定的属性失败，搜索过 `obj`、`Class` 和超类之后调用。表达式 `obj.no_such_attr`、`getattr(obj, 'no_such_attr')` 和 `hasattr(obj, 'no_such_attr')` 可能会触发 `Class.__getattr__(obj, 'no_such_attr')` 方法，但是，仅当在 `obj`、`Class` 和超类中找不到指定的属性时才会触发。

`__getattribute__(self, name)`

尝试获取指定的属性时总会调用这个方法，不过，寻找的属性是特殊属性或特殊方法时除外。点号与 `getattr` 和 `hasattr` 内置函数会触发这个方法。调用 `__getattribute__` 方法且抛出 `AttributeError` 异常时，才会调用 `__getattr__` 方法。为了在获取 `obj` 实例的属性时不导致无限递归，`__getattribute__` 方法的实现要使用 `super().__getattribute__(obj, name)`。

`__setattr__(self, name, value)`

尝试设置指定的属性时总会调用这个方法。点号和 `setattr` 内置函数会触发这个方法。例如，`obj.attr = 42` 和 `setattr(obj, 'attr', 42)` 都会触发 `Class.__setattr__(obj, ‘attr’, 42)` 方法。

> 其实，特殊方法 `__getattribute__` 和 `__setattr__` 不管怎样都会调用，几乎会影响每一次属性存取，因此比 `__getattr__` 方法（只处理不存在的属性名）更难正确使用。与定义这些特殊方法相比，使用特性或描述符相对不易出错。

我们对特性、特殊方法和其他动态属性编程技术的讨论到此结束。

19.7　本章小结

本章的话题是动态属性编程。我们首先举了几个实例，定义了几个简单的类，简化处理 JSON 数据源的方式。第一个示例是 FrozenJSON 类，把嵌套的字典和列表转换成嵌套的 FrozenJSON 实例和实例列表。FrozenJSON 类的代码展示了如何使用特殊的 __getattr__ 方法在读取属性时即时转换数据结构。FrozenJSON 类的最后一版展示了如何使用 __new__ 构造方法把一个类转换成一个灵活的对象工厂函数，不受实例本身的限制。

然后，我们把 JSON 源转换成一个 shelve.Shelf 数据库，把序列化的 Record 实例存在里面。第 1 版 Record 类只有几行代码，介绍了“集束”惯用法：使用传给 __init__ 方法的关键字参数，调用 self.__dict__.update(**kwargs) 构建任意属性。这个示例的第 2 版对 Record 类做了扩展：一个是 DbRecord 类，集成数据库操作；另一个是 Event 类，通过特性自动获取所链接的记录。

接着，本章讨论了特性。我们定义的 LineItem 类中有个特性，确保 weight 属性的值不能是对业务没有意义的负数或零。然后，我们深入说明了特性的句法和语义。随后，创建了一个特性工厂函数，在不定义多个读值方法和设值方法的前提下，对 weight 和 price 属性做相同的验证。那个特性工厂函数用到了几个精妙的概念，例如闭包和被特性覆盖的实例属性，提供了优雅的通用方案，代码行数与用手工编码的特性来验证单个属性的一样多。

最后，我们简要说明了如何使用特性处理删除属性的操作，随后概览了 Python 核心语言为支持属性元编程而提供的重要的特殊属性、内置函数和特殊方法。

19.8 延伸阅读

属性处理和内置的内省函数的官方文档在 Python 标准库文档的第 2 章中，题为“Built-in Functions”（https://docs.python.org/3/library/functions.html）。相关的特殊方法和特殊的 `__slots__` 属性在 Python 语言参考手册中的“3.3.2. Customizing attribute access”一节（https://docs.python.org/3/reference/datamodel.html#customizing-attribute-access）里说明。调用特殊方法会跳过实例的语意原因在“3.3.9. Special method lookup”一节（https://docs.python.org/3/reference/datamodel.html#special-method-lookup）中说明。在 Python 标准库文档的第 4 章“Built-in Types”里，“4.13. Special Attributes”一节（https://docs.python.org/3/library/stdtypes.html#special-attributes）说明了 `__class__` 和 `__dict__` 属性。

David Beazley 与 Brian K. Jones 的《Python Cookbook（第 3 版）中文版》一书中有几个诀窍涉及本章的话题，不过我要重点提出三个：“8.8 在子类中扩展属性”，解决了在继承自超类的特性中覆盖方法这个棘手问题；“8.15 委托属性的访问”，实现了一个代理类，展示了本书 19.6.3 节所列的大多数特殊方法；还有出色的“9.21 避免出现重复的属性方法”一节，示例 19-24 中定义的特性工厂函数就以那一节为基础。

Alex Martelli 写的《Python 技术手册（第 2 版）》只涵盖了 Python 2.5，不过基础知识也适用于 Python 3。他写书的风格严谨而客观，讲到特性时，只用了 3 页，但这是由于那本书采用了符合逻辑的行文方式：之前的 15 页已经对 Python 的类做了详尽的说明，包括描述符，而特性就是使用描述符实现的。因此讲到特性时，他可以在 3 页的篇幅中发表很多见解，例如本章开篇引用的那句话。

本章开头引用的统一访问原则定义出自 Bertrand Meyer 的优秀著作 *Object-Oriented Software Construction, Second Edition*（Prentice-Hall 出版社）。这本书超过 1250 页，我承认我没有读完，不过前六章对面向对象分析和设计相关概念的介绍是我见过最好的之一，第 11 章介绍了契约式设计（Meyer 发明了这种设计方法，创造了这个术语），第 35

章阐述了他对重要的面向对象语言的评价，包括 Simula、Smalltalk、CLOS（Lisp 的面向对象扩展）、 Objective-C、C++ 和 Java，还对其他语言做了简要评述。他还发明了伪伪代码（pseudo- pseudocode），直到那本书的最后一页他才披露，全书用于编写伪代码的句法其实出自 Eiffel 语言。

杂谈

站在美学的角度来看，Meyer 提出的统一访问原则（Unifrom Access Principle，喜欢简称的人有时称之为UAP）很吸引人。作为使用 API 的程序员，我不应该关心 `coconut.price` 只是获取数据属性还是执行计算。但是，作为消费者和公民，我应该关心：在电子商务发达的今天，`coconut.price` 的值通常取决于这个问题由谁提出，因此它绝不仅仅是个数据属性。其实，如果查询来自网店外部（例如比价引擎），价格通常会低一些。显然，这对喜欢浏览特定网店的忠实消费者来说，利益受到了损害。但是我不同意。

前一段离题了，可是却提出了与编程有关的问题：虽然统一访问原则在理想的世界中完全合理，但在现实中，API 的用户可能需要知道读取 `coconut.price` 是否太耗资源或时间。Ward Cunningham 的维基（http://c2.com/cgi/wiki?WelcomeVisitors）对软件工程方面的话题有很多独到的见解，他对统一访问原则的功过也做了富有洞察力的论述（http://c2.com/cgi/wiki?UniformAccessPrinciple）。

在面向对象编程语言中，是否遵守统一访问原则通常体现在句法上：究竟是读取公开的数据属性，还是调用读值方法和设值方法。

Smalltalk 和 Ruby 使用简单而优雅的方式解决这个问题：根本不支持公开的数据属性。在这两门语言中，所有实例属性都是私有的，因此必须通过方法来存取。不过，这两门语言的句法把这个过程变得毫不费力：在 Ruby 中，`coconut.price` 会调用读值方法 `price`；在 Smalltalk 中，只需使用 `coconut price`。

Java 采用的是另一种方式，让程序员在四种访问级别修饰符中选择。[15] 不过，普通大众并不认同 Java 设计者制定的这种句法。Java 世界的人都认为，属性应该是私有的，但是每一次都要写出 `private`，因为这不是默认的访问级别。如果所有属性都是私有的，那么从类外部访问属性就必须使用存取方法。Java IDE 提供了

自动生成存取方法的快捷方式。但是，六个月后不得不阅读代码时，IDE 没有多大帮助。我们要在众多什么也没做的存取方法中找出所需的那一个，添加实现某些业务逻辑所需的值。

Alex Martelli 把存取方法称为“愚蠢的惯用法”，这道出了 Python 社区中大多数人的心声。他举了下面两个例子，外观差异很大，但是作用相同：[16]

```
someInstance.widgetCounter += 1
# 而不用……
someInstance.setWidgetCounter(someInstance.getWidgetCounter() + 1)
```

设计 API 时，我有时会想，能否把没有参数（除了 `self`）、返回一个值（除了 `None`）的纯函数（即没有副作用）替换成只读特性。在本章中，`LineItem.subtotal` 方法（如示例 19-23 所示）就可以替换成只读特性。当然，用于修改对象的方法（如 `my_list.clear()`）不在此列。把这样的方法变成特性是个糟糕的想法，因为直接访问 `my_list.clear` 就会删除列表中的内容。

在 GPIO 库 Pingo.io（http://www.pingo.io/docs/，3.4.2 节提过）中，多数用户级别的 API 都基于特性实现。例如，为了读取模拟针脚的当前值，用户要编写 `pin.value`；为了设置数字针脚的模式，要写成 `pin.mode = OUT`。在背后，读取模拟针脚的值或设置数字针脚的模式可能涉及大量代码，这取决于具体的主板驱动。我们决定在 Pingo 中使用特性，是因为我们想让 API 用起来舒服，即便是在 iPython Notebook（http://ipython.org/notebook.html）等交互环境中也是如此，而且我们觉得 `pin.mode = OUT` 看起来和输入起来都比 `pin.set_mode(OUT)` 容易。

我觉得 Smalltalk 和 Ruby 的处理方式很简洁，但也认为 Python 的处理方式比 Java 更合理。一开始，我们可以从简单的方式入手，把数据成员定义为公开的属性，因为我们知道这些属性可以使用特性（或下一章讨论的描述符）来包装。

`__new__` 方法比 new 运算符好

在 Python 中还有一处体现了统一访问原则（或者它的变体）：函数

调用和对象实例化使用相同的句法——my_obj = foo()，其中 foo 是类或其他可调用的对象。

受 C++ 句法影响的其他语言提供了 new 运算符，致使实例化不像是调用。大多数时候，API 的用户不关心 foo 是函数还是类。直到最近，我才意识到，property 是个函数。在常规的用法中，这没什么区别。

把构造方法替换成工厂方法有很多充足的理由。[17] 一个重要的原因是，通过返回之前构建的实例，限制实例的数量（体现了单例模式）。有个相关的功能是，缓存构建过程开销大的对象。此外，有时便于根据指定的参数返回不同类型的对象。

定义构造方法较为简单；提供工厂方法虽然增加了灵活性，但是要编写更多的代码。在有 new 运算符的语言中，API 的设计者必须提前决定：究竟是坚持使用简单的构造方法，还是投入工厂方法的怀抱。如果一开始选择错了，那么修正的代价可能很大——这一切都因为 new 是运算符。

有时可能更适合走另一条路，把简单的函数换成类。

在 Python 中，很多情况下类和函数可以互换。这不仅是因为 Python 没有 new 运算符，还因为有特殊的 __new__ 方法，可以把类变成工厂方法，生成不同类型的对象（如 19.1.3 节所述），或者返回事先构建好的实例，而不是每次都创建一个新实例。

如果“PEP 8—Style Guide for Python Code”（https://www.python.org/dev/peps/pep-0008/#class-names）不推荐类名使用驼峰式（CamelCase），那么函数与类的对偶性更易于使用。不过，标准库中有很多类的名称是小写的（例如 property、str、defaultdict，等等）。因此，使用小写的类名可能是个特色，而不是缺陷。但是，不管怎么看，Python 标准库在类名大小写上的不一致会导致可用性问题。

虽然调用函数与调用类没有区别，但是最好知道哪个是哪个，因为类还有一个功能：子类化。因此，我编写的每个类都使用驼峰式名称，而且希望 Python 标准库中的所有类也使用这一约定。我在盯着你呢，collections.OrderedDict 和

collections.defaultdict。

[15]包括没有名称的默认级别，Java 教程（http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html）称其为“包级私有”。

[16]《Python 技术手册（第 2 版）》第 101 页。

[17]我将要提到的原因出自 Jonathan Amsterdam 发布在 Dr. Dobbs Journal 中的一篇文章，题为“Java's new Considered Harmful”（http://www.drdobbs.com/javas-new-considered-harmful/184405016），以及 Joshua Bloch 写的获奖图书 *Effective Java* 中的第一条，“考虑用静态工厂方法代替构造函数”。

第 20 章　属性描述符

> 学会描述符之后，不仅有更多的工具集可用，还会对 Python 的运作方式有更深入的理解，并由衷赞叹 Python 设计的优雅。[1]
>
> ——Raymond Hettinger
> Python 核心开发者和专家

[1]摘自 Raymond Hettinger 写的“Descriptor HowTo Guide”（https://docs.python.org/3/howto/descriptor.html）。

描述符是对多个属性运用相同存取逻辑的一种方式。例如，Django ORM 和 SQL Alchemy 等 ORM 中的字段类型是描述符，把数据库记录中字段里的数据与 Python 对象的属性对应起来。

描述符是实现了特定协议的类，这个协议包括 `__get__`、`__set__` 和 `__delete__` 方法。`property` 类实现了完整的描述符协议。通常，可以只实现部分协议。其实，我们在真实的代码中见到的大多数描述符只实现了 `__get__` 和 `__set__` 方法，还有很多只实现了其中的一个。

描述符是 Python 的独有特征，不仅在应用层中使用，在语言的基础设施中也有用到。除了特性之外，使用描述符的 Python 功能还有方法及 `classmethod` 和 `staticmethod` 装饰器。理解描述符是精通 Python 的关键。本章的话题就是描述符。

20.1　描述符示例：验证属性

如 19.4 节所示，特性工厂函数借助函数式编程模式避免重复编写读值方法和设值方法。特性工厂函数是高阶函数，在闭包中存储 `storage_name` 等设置，由参数决定创建哪些存取函数，再使用存取函数构建一个特性实例。解决这种问题的面向对象方式是描述符类。

这里继续 19.4 节的 `LineItem` 系列示例，把 `quantity` 特性工厂函数重构成 `Quantity` 描述符类。

20.1.1　`LineItem`类第3版：一个简单的描述符

实现了 `__get__`、`__set__` 或 `__delete__` 方法的类是描述符。描述符的用法是，创建一个实例，作为另一个类的类属性。

我们将定义一个 `Quantity` 描述符，`LineItem` 类会用到两个 `Quantity` 实例：一个用于管理 `weight` 属性，另一个用于管理 `price` 属性。示意图有助于理解，如图 20-1 所示。

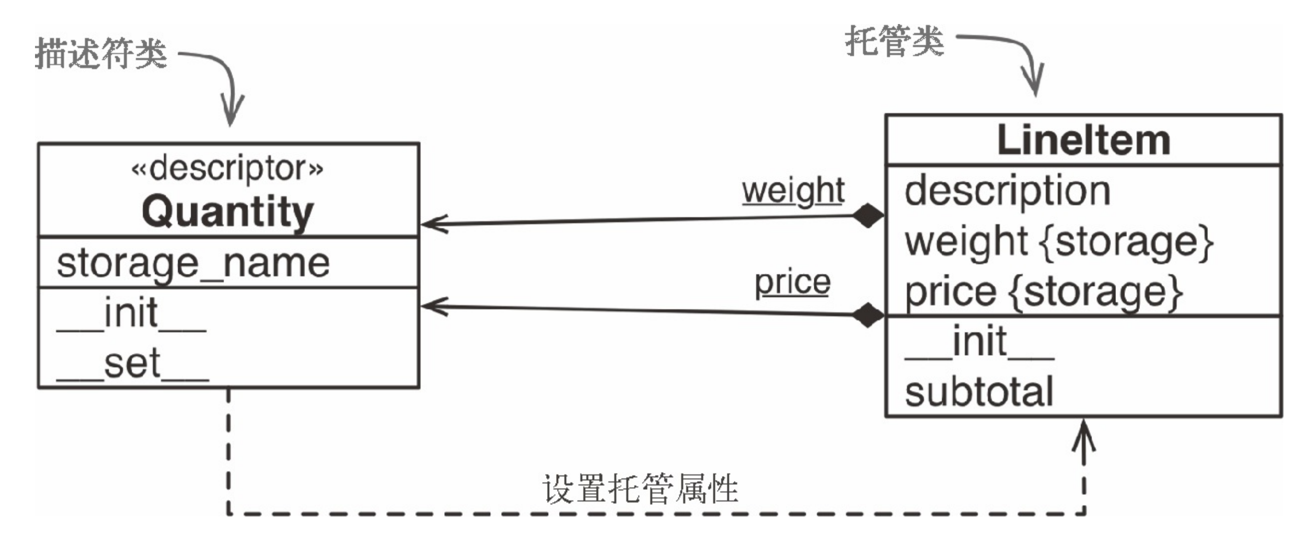


图 **20-1**：**`LineItem`** 类的 **UML** 示意图，用到了名为 **`Quantity`** 的描述符类。**UML** 示意图中带下划线的属性是类属性。注意，**`weight`** 和 **`price`** 是依附在 **`LineItem`** 类上的 **`Quantity`** 类的实例，不过 **`LineItem`** 实例也有自己的 **`weight`** 和 **`price`** 属性，存储着相应的值

注意，在图 20-1 中，“weight”这个词出现了两次，因为其实有两个不同的属性都叫 `weight`：一个是 `LineItem` 的类属性，另一个是各个 `LineItem` 对象的实例属性。`price` 也是如此。

从现在开始，我会使用下述定义。

描述符类

　　实现描述符协议的类。在图 20-1 中，是 `Quantity` 类。

托管类

　　把描述符实例声明为类属性的类——图 20-1 中的 `LineItem` 类。

描述符实例

　　描述符类的各个实例，声明为托管类的类属性。在图 20-1 中，各个描述符实例使用箭头和带下划线的名称表示（在 UML 中，下划线表示类属性）。与黑色菱形接触的 `LineItem` 类包含描述符实例。

托管实例

　　托管类的实例。在这个示例中，`LineItem` 实例是托管实例（没在类图中展示）。

储存属性

　　托管实例中存储自身托管属性的属性。在图 20-1 中，`LineItem` 实例的 `weight` 和 `price` 属性是储存属性。这种属性与描述符属性不同，描述符属性都是类属性。

托管属性

　　托管类中由描述符实例处理的公开属性，值存储在储存属性中。也就是说，描述符实例和储存属性为托管属性建立了基础。

`Quantity` 实例是 `LineItem` 类的类属性，这一点一定要理解。图 20-2 中的机器和小怪兽强调了这个关键点。

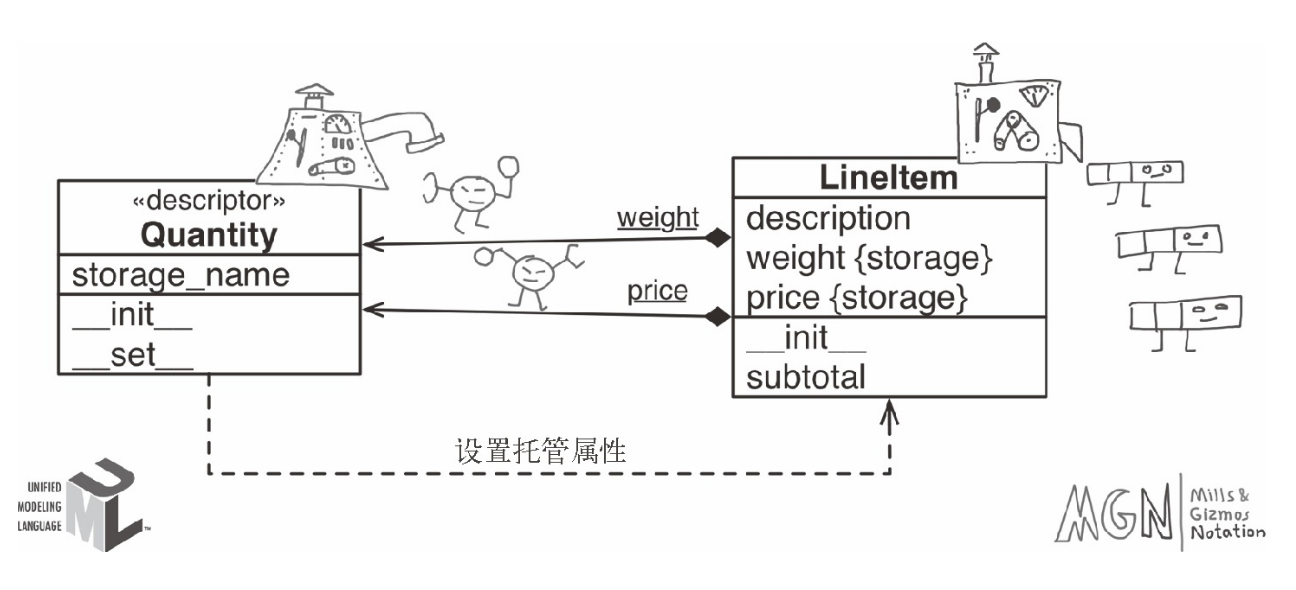


图 **20-2**：带有 **MGN**（**Mills & Gizmos Notation**，机器和小怪兽图示法）注解的 **UML** 类图：类是机器，用于生产小怪兽（实例）。**`Quantity`** 机器生产了两个圆头的小怪兽，依附到 **`LineItem`** 机器上，即 **`weight`** 和 **`price`**。**`LineItem`** 机器生产方头的小怪兽，有自己的 **`weight`** 和 **`price`** 属性，存储着相应的值

机器和小怪兽图示法介绍

我以前经常使用 UML 解说描述符，但是后来发现 UML 无法很好地展现类与实例之间的关系，例如托管类与描述符实例之间的关系。[2] 所以，我自己发明了一门“语言”——机器和小怪兽图示法（Mills & Gizmos Notation，MGN），使用它注解 UML 示意图。

MGN 的目的是明确区分类和实例。如图 20-3 所示。在 MGN 中，类画成“机器”，这是一种复杂的设备，用于生产小怪兽。类（机器）都是有操控杆和刻度盘的设备。小怪兽是实例，外观更简洁。小怪兽与生产它的机器具有相同的颜色。

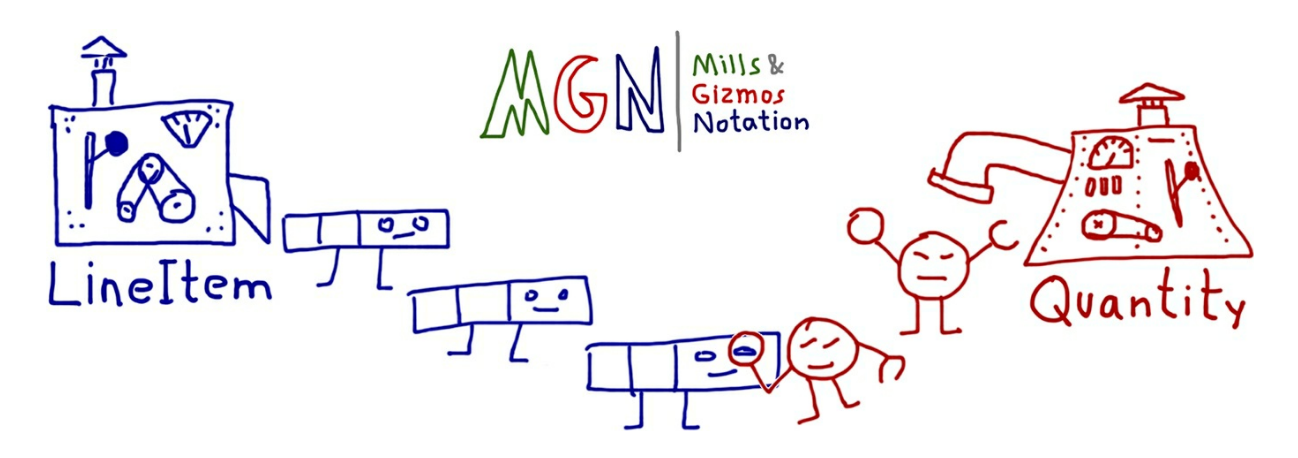


图 **20-3**：**MGN** 简图表示，**LineItem** 类生产了三个实例，**Quantity** 类生产了两个实例。其中一个 **Quantity** 实例从一个 **LineItem** 实例中获取存储的值

在这个示例中，我把 `LineItem` 实例画成表格中的行，各有三个单元格，表示三个属性（`description`、`weight` 和 `price`）。`Quantity` 实例是描述符，因此有个放大镜，用于获取值（`__get__`），以及一个手抓，用于设置值（`__set__`）。讲到元类时，你会感谢我画了这些涂鸦。

> [2]在 UML 类图中，类和实例都画成方框。虽然视觉上有区别，但是因为类图中很少出现实例，所以开发者可能认不出。

先把涂鸦放在一边，来看代码：示例 20-1 是 `Quantity` 描述符类和新版 `LineItem` 类，用到两个 `Quantity` 实例。

示例 **20-1**　bulkfood_v3.py：使用 `Quantity` 描述符管理 `LineItem` 的属性

```
class Quantity:  ❶

    def __init__(self, storage_name):
        self.storage_name = storage_name  ❷

    def __set__(self, instance, value):  ❸
        if value > 0:
            instance.__dict__[self.storage_name] = value  ❹
        else:
            raise ValueError('value must be > 0')
```

```
class LineItem:
    weight = Quantity('weight')  ❺
    price = Quantity('price')  ❻

    def __init__(self, description, weight, price):  ❼
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ 描述符基于协议实现，无需创建子类。

❷ Quantity 实例有个 storage_name 属性，这是托管实例中存储值的属性的名称。

❸ 尝试为托管属性赋值时，会调用 __set__ 方法。这里，self 是描述符实例（即 LineItem.weight 或 LineItem.price），instance 是托管实例（LineItem 实例），value 是要设定的值。

❹ 这里，必须直接处理托管实例的 __dict__ 属性；如果使用内置的 setattr 函数，会再次触发 __set__ 方法，导致无限递归。

❺ 第一个描述符实例绑定给 weight 属性。

❻ 第二个描述符实例绑定给 price 属性。

❼ 类定义体中余下的代码与 bulkfood_v1.py 脚本（见示例 19-15）中的代码一样简洁。

在示例 20-1 中，各个托管属性的名称与储存属性一样，而且读值方法不需要特殊的逻辑，所以 Quantity 类不需要定义 __get__ 方法。

示例 20-1 中的代码会像预期那样运作，禁止以 0 美元销售松露：[3]

> [3]一磅白松露价值几千美元。留个练习给有进取心的读者：不准以 0.01 美元的价格销售松露。我认识一个人，他以 18 美元买到了价值 1800 美元的统计学百科全书，因为那个网店（不是亚马逊）有漏洞。

```
>>> truffle = LineItem('White truffle', 100, 0)
Traceback (most recent call last):
    ...
ValueError: value must be > 0
```

编写 `__set__` 方法时，要记住 `self` 和 `instance` 参数的意思：`self` 是描述符实例，`instance` 是托管实例。管理实例属性的描述符应该把值存储在托管实例中。因此，Python 才为描述符中的那个方法提供了 `instance` 参数。

你可能想把各个托管属性的值直接存在描述符实例中，但是这种做法是错误的。也就是说，在 `__set__` 方法中，应该像下面这样写：

```
    instance.__dict__[self.storage_name] = value
```

而不能试图使用下面这种错误的写法：

```
    self.__dict__[self.storage_name] = value
```

为了理解错误的原因，可以想想 `__set__` 方法前两个参数（`self` 和 `instance`）的意思。这里，`self` 是描述符实例，它其实是托管类的类属性。同一时刻，内存中可能有几千个 `LineItem` 实例，不过只会有两个描述符实例：`LineItem.weight` 和 `LineItem.price`。因此，存储在描述符实例中的数据，其实会变成 `LineItem` 类的类属性，从而由全部 `LineItem` 实例共享。

示例 20-1 有个缺点，在托管类的定义体中实例化描述符时要重复输入属性的名称。如果 `LineItem` 类能像下面这样声明就好了：

```
class LineItem:
    weight = Quantity()
    price = Quantity()

    # 余下的方法与之前一样
```

可问题是，正如第 8 章说过的，赋值语句右手边的表达式先执行，而此时变量还不存在。Quantity() 表达式计算的结果是创建描述符实例，而此时 Quantity 类中的代码无法猜出要把描述符绑定给哪个变量（例如 weight 或 price）。

因此，示例 20-1 必须明确指明各个 Quantity 实例的名称。这样不仅麻烦，还很危险：如果程序员直接复制粘贴代码而忘了编辑名称，比如写成 price = Quantity('weight')，那么程序的行为会大错特错，设置 price 的值时会覆盖 weight 的值。

下一节会介绍一个不太优雅但是可行的方案，解决这个重复输入名称的问题。更好的解决方案是使用类装饰器或元类，等到第 21 章再介绍。

20.1.2　LineItem类第4版：自动获取储存属性的名称

为了避免在描述符声明语句中重复输入属性名，我们将为每个 Quantity 实例的 storage_name 属性生成一个独一无二的字符串。图 20-4 是更新后的 Quantity 和 LineItem 类的 UML 类图。

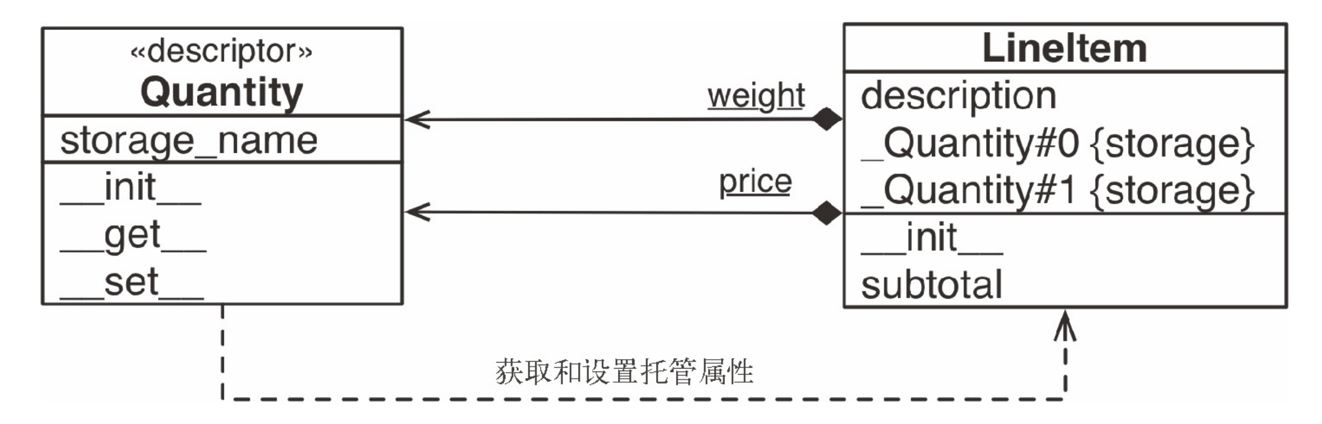


图 **20-4**：示例 **20-2** 的 **UML** 类图。现在，**Quantity** 类既有 **__get__** 方法，也有 **__set__** 方法；**LineItem** 实例中储存属性的名称是生成的，**_Quantity#0** 和 **_Quantity#1**

为了生成 storage_name，我们以 '_Quantity#' 为前缀，然后在后面拼接一个整数：Quantity.__counter 类属性的当前值，每次把一个新的 Quantity 描述符实例依附到类上，都会递增这个值。在前缀中使用井号能避免 storage_name 与用户使用点号创建的属性冲突，因为

nutmeg._Quantity#0 是无效的 Python 句法。但是，内置的 getattr 和 setattr 函数可以使用这种“无效的”标识符获取和设置属性，此外也可以直接处理实例属性 __dict__。示例 20-2 是新的实现。

示例 **20-2**　bulkfood_v4.py：每个 Quantity 描述符都有独一无二的 storage_name

```
class Quantity:
    __counter = 0  ❶

    def __init__(self):
        cls = self.__class__  ❷
        prefix = cls.__name__
        index = cls.__counter
        self.storage_name = '_{}#{}'.format(prefix, index)  ❸
        cls.__counter += 1  ❹

    def __get__(self, instance, owner):  ❺
        return getattr(instance, self.storage_name)  ❻

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.storage_name, value)  ❼
        else:
            raise ValueError('value must be > 0')


class LineItem:
    weight = Quantity()  ❽
    price = Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ __counter 是 Quantity 类的类属性，统计 Quantity 实例的数量。

❷ cls 是 Quantity 类的引用。

❸ 每个描述符实例的 `storage_name` 属性都是独一无二的，因为其值由描述符类的名称和 `__counter` 属性的当前值构成（例如，`_Quantity#0`）。

❹ 递增 `__counter` 属性的值。

❺ 我们要实现 `__get__` 方法，因为托管属性的名称与 `storage_name` 不同。稍后会说明 `owner` 参数。

❻ 使用内置的 `getattr` 函数从 `instance` 中获取储存属性的值。

❼ 使用内置的 `setattr` 函数把值存储在 `instance` 中。

❽ 现在，不用把托管属性的名称传给 `Quantity` 构造方法。这是这一版的目标。

这里可以使用内置的高阶函数 `getattr` 和 `setattr` 存取值，无需使用 `instance.__dict__`，因为托管属性和储存属性的名称不同，所以把储存属性传给 `getattr` 函数不会触发描述符，不会像示例 20-1 那样出现无限递归。

测试 bulkfood_v4.py 脚本之后你会发现，`weight` 和 `price` 描述符能按预期使用，而且储存属性也能直接读取——这对调试有帮助：

```
>>> from bulkfood_v4 import LineItem
>>> coconuts = LineItem('Brazilian coconut', 20, 17.95)
>>> coconuts.weight, coconuts.price
(20, 17.95)
>>> getattr(coconuts, '_Quantity#0'), getattr(coconuts, '_Quantity#1')
(20, 17.95)
```

如果想使用 Python 矫正名称的约定方式（例如 `_LineItem__quantity0`），要知道托管类（即 `LineItem`）的名称，可是，解释器要先运行类的定义体才能构建类，因此创建描述符实例时得不到那个信息。不过，对这个示例来说，为了防止不小心被子类覆盖，不用包含托管类的名称，因为每次实例化新的描述符，描述符类的 `__counter` 属性都会递增，从而确保每个托管类的每个储存属性的名称都是独一无二的。

注意，__get__ 方法有三个参数：self、instance 和 owner。owner 参数是托管类（如 LineItem）的引用，通过描述符从托管类中获取属性时用得到。如果使用 LineItem.weight 从类中获取托管属性（以 weight 为例），描述符的 __get__ 方法接收到的 instance 参数值是 None。因此，下述控制台会话才会抛出 AttributeError 异常：

```
>>> from bulkfood_v4 import LineItem
>>> LineItem.weight
Traceback (most recent call last):
  ...
  File ".../descriptors/bulkfood_v4.py", line 54, in __get__
    return getattr(instance, self.storage_name)
AttributeError: 'NoneType' object has no attribute '_Quantity#0'
```

抛出 AttributeError 异常是实现 __get__ 方法的方式之一，如果选择这么做，应该修改错误消息，去掉令人困惑的 NoneType 和 _Quantity#0，这是实现细节。把错误消息改成"'LineItem' class has no such attribute" 更好。最好能给出缺少的属性名，但是在这个示例中，描述符不知道托管属性的名称，因此目前只能做到这样。

此外，为了给用户提供内省和其他元编程技术支持，通过类访问托管属性时，最好让 __get__ 方法返回描述符实例。示例 20-3 对示例 20-2 做了小幅改动，为 Quantity.__get__ 方法添加了一些逻辑。

示例 **20-3**　bulkfood_v4b.py（只列出部分代码）：通过托管类调用时，__get__ 方法返回描述符的引用

```
class Quantity:
    __counter = 0

    def __init__(self):
        cls = self.__class__
        prefix = cls.__name__
        index = cls.__counter
        self.storage_name = '_{}#{}'.format(prefix, index)
        cls.__counter += 1

    def __get__(self, instance, owner):
        if instance is None:
            return self  ❶
```

```
        else:
            return getattr(instance, self.storage_name)  ❷

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.storage_name, value)
        else:
            raise ValueError('value must be > 0')
```

❶ 如果不是通过实例调用，返回描述符自身。

❷ 否则，像之前一样，返回托管属性的值。

测试示例 20-3，会看到如下结果：

```
>>> from bulkfood_v4b import LineItem
>>> LineItem.price
<bulkfood_v4b.Quantity object at 0x100721be0>
>>> br_nuts = LineItem('Brazil nuts', 10, 34.95)
>>> br_nuts.price
34.95
```

看着示例 20-2，你可能觉得就为了管理几个属性而编写这么多代码不值得，但是要知道，描述符逻辑现在被抽象到单独的代码单元（`Quantity` 类）中了。通常，我们不会在使用描述符的模块中定义描述符，而是在一个单独的实用工具模块中定义，以便在整个应用中使用——如果开发的是框架，甚至会在多个应用中使用。

了解这一点之后就可推知，示例 20-4 是描述符的常规用法。

示例 **20-4**　bulkfood_v4c.py：整洁的 `LineItem` 类；`Quantity` 描述符类现在位于导入的 `model_v4c` 模块中

```
import model_v4c as model  ❶


class LineItem:
    weight = model.Quantity()  ❷
    price = model.Quantity()

    def __init__(self, description, weight, price):
```

```
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ 导入 `model_v4c` 模块，指定一个更友好的名称。

❷ 使用 `model.Quantity` 描述符。

Django 用户会发现，示例 20-4 非常像模型定义。这不是巧合：Django 模型的字段就是描述符。

> 就目前的实现来说，`Quantity` 描述符能出色地完成任务。它唯一的缺点是，储存属性的名称是生成的（如 `_Quantity#0`），导致用户难以调试。但这是不得已而为之，如果想自动把储存属性的名称设成与托管属性的名称类似，需要用到类装饰器或元类，而这两个话题到第 21 章才会讨论。

描述符在类中定义，因此可以利用继承重用部分代码来创建新描述符。下一节会这么做。

> 特性工厂函数与描述符类比较
>
> 特性工厂函数若想实现示例 20-2 中增强的描述符类并不难，只需在示例 19-24 的基础上添加几行代码。`__counter` 变量的实现方式是个难点，不过我们可以把它定义成工厂函数对象的属性，以便在多次调用之间持续存在，如示例 20-5 所示。
>
> 示例 **20-5**　bulkfood_v4prop.py：使用特性工厂函数实现与示例 20-2 中的描述符类相同的功能
>
> ```
> def quantity():  ❶
>     try:
>         quantity.counter += 1  ❷
>     except AttributeError:
>         quantity.counter = 0  ❸
> ```

```
    storage_name = '_{}:{}'.format('quantity', quantity.counter)  ❹

    def qty_getter(instance):  ❺
        return getattr(instance, storage_name)

     def qty_setter(instance, value):
        if value > 0:
            setattr(instance, storage_name, value)
        else:
            raise ValueError('value must be > 0')

    return property(qty_getter, qty_setter)
```

❶ 没有 `storage_name` 参数。

❷ 不能依靠类属性在多次调用之间共享 `counter`，因此把它定义为 `quantity` 函数自身的属性。

❸ 如果 `quantity.counter` 属性未定义，把值设为 `0`。

❹ 我们也没有实例变量，因此创建一个局部变量 `storage_name`，借助闭包保持它的值，供后面的 `qty_getter` 和 `qty_setter` 函数使用。

❺ 余下的代码与示例 19-24 一样，不过这里可以使用内置的 `getattr` 和 `setattr` 函数，而不用处理 `instance.__dict__` 属性。

那么，你喜欢哪个？示例 20-2 还是示例 20-5 ？

我喜欢描述符类那种方式，主要有下列两个原因。

- 描述符类可以使用子类扩展；若想重用工厂函数中的代码，除了复制粘贴，很难有其他方法。

- 与示例 20-5 中使用函数属性和闭包保持状态相比，在类属性和实例属性中保持状态更易于理解。

此外，解说示例 20-5 时，我没有画机器和小怪兽的动力。特性工

厂函数的代码不依赖奇怪的对象关系，而描述符的方法中有名为 self 和 instance 的参数，表明里面涉及奇怪的对象关系。

总之，从某种程度上来讲，特性工厂函数模式较简单，可是描述符类方式更易扩展，而且应用也更广泛。

20.1.3　LineItem类第5版：一种新型描述符

我们虚构的有机食物网店遇到一个问题：不知怎么回事儿，有个商品的描述信息为空，导致无法下订单。为了避免出现这个问题，我们要再创建一个描述符，NonBlank。在设计 NonBlank 的过程中，我们发现，它与 Quantity 描述符很像，只是验证逻辑不同。

回想 Quantity 的功能，我们注意到它做了两件不同的事：管理托管实例中的储存属性，以及验证用于设置那两个属性的值。由此可知，我们可以重构，并创建两个基类。

AutoStorage

　　自动管理储存属性的描述符类。

Validated

　　扩展 AutoStorage 类的抽象子类，覆盖 __set__ 方法，调用必须由子类实现的 validate 方法。

我们稍后会重写 Quantity 类，并实现 NonBlank，让它继承 Validated 类，只编写 validate 方法。类之间的关系见图 20-5。

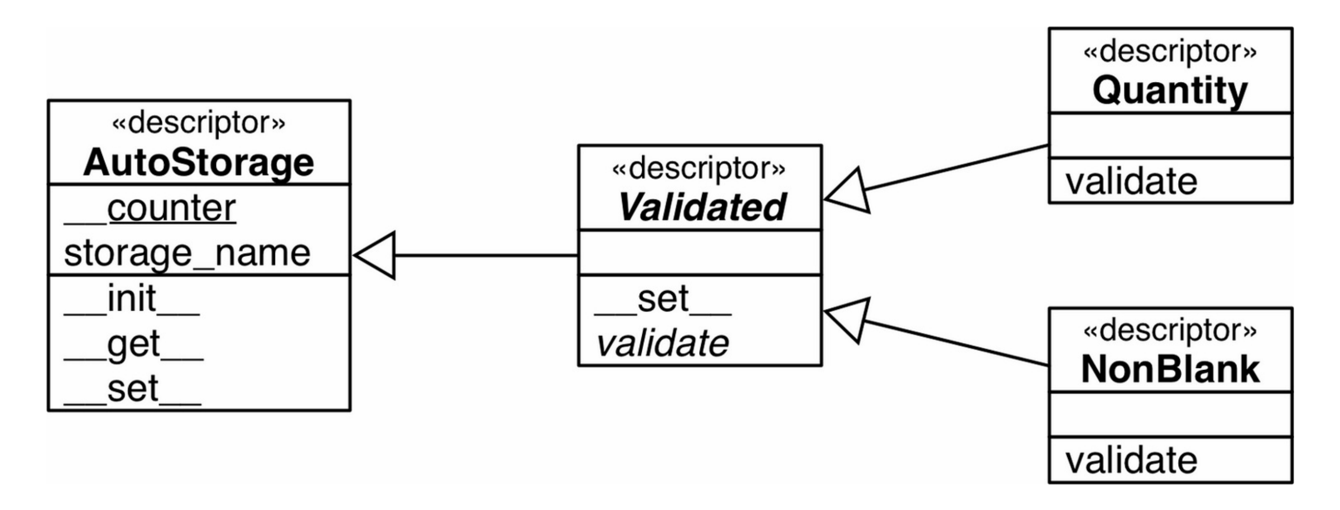


图 **20-5**：几个描述符类的层次结构。**AutoStorage** 基类负责自动存储属性；**Validated** 类做验证，把职责委托给抽象方法 **validate**；**Quantity** 和 **NonBlank** 是 **Validated** 的具体子类

`Validated`、`Quantity` 和 `NonBlank` 三个类之间的关系体现了模板方法设计模式。具体而言，`Validated.__set__` 方法正是 Gamma 等四人所描述的模板方法的例证：

> 一个模板方法用一些抽象的操作定义一个算法，而子类将重定义这些操作以提供具体的行为。[4]

> [4]《设计模式：可复用面向对象软件的基础》第 215 页。

这里，抽象的操作是验证。示例 20-6 列出图 20-5 中各个类的实现。

示例 **20-6**　model_v5.py：重构后的描述符类[5]

> [5]因为 20.5 节有文档字符串的截图，为了保持一致，所以这里的文档字符串不翻译。——译者注

```
import abc


class AutoStorage:  ❶
    __counter = 0

    def __init__(self):
        cls = self.__class__
```

```
        prefix = cls.__name__
        index = cls.__counter
        self.storage_name = '_{}#{}'.format(prefix, index)
        cls.__counter += 1

    def __get__(self, instance, owner):
        if instance is None:
            return self
        else:
            return getattr(instance, self.storage_name)

    def __set__(self, instance, value):
        setattr(instance, self.storage_name, value)  ❷


class Validated(abc.ABC, AutoStorage):  ❸

    def __set__(self, instance, value):
        value = self.validate(instance, value)  ❹
        super().__set__(instance, value)  ❺

    @abc.abstractmethod
    def validate(self, instance, value):  ❻
        """return validated value or raise ValueError"""


class Quantity(Validated):  ❼
    """a number greater than zero"""

    def validate(self, instance, value):
        if value <= 0:
            raise ValueError('value must be > 0')
        return value


class NonBlank(Validated):
    """a string with at least one non-space character"""

    def validate(self, instance, value):
        value = value.strip()
        if len(value) == 0:
            raise ValueError('value cannot be empty or blank')
        return value  ❽
```

❶ AutoStorage 类提供了之前 Quantity 描述符的大部分功能……

❷ ……验证除外。

❸ `Validated` 是抽象类，不过也继承自 `AutoStorage` 类。

❹ `__set__` 方法把验证操作委托给 `validate` 方法……

❺ ……然后把返回的 `value` 传给超类的 `__set__` 方法，存储值。

❻ 在这个类中，`validate` 是抽象方法。

❼ `Quantity` 和 `NonBlank` 都继承自 `Validated` 类。

❽ 要求具体的 `validate` 方法返回验证后的值，借机可以清理、转换或规范化接收的数据。这里，我们把 `value` 首尾的空白去掉，然后将其返回。

model_v5.py 脚本的用户不需要知道全部细节。用户只需知道，他们可以使用 `Quantity` 和 `NonBlank` 自动验证实例属性。参见示例 20-7 中的最新版 `LineItem` 类。

示例 **20-7**　bulkfood_v5.py：使用 `Quantity` 和 `NonBlank` 描述符的 `LineItem` 类

```
import model_v5 as model  ❶


class LineItem:
    description = model.NonBlank()  ❷
    weight = model.Quantity()
    price = model.Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ 导入 `model_v5` 模块，指定一个更友好的名称。

❷ 使用 `model.NonBlank` 描述符。其余的代码没变。

本章所举的几个 `LineItem` 示例演示了描述符的典型用途——管理数据属性。这种描述符也叫覆盖型描述符，因为描述符的 `__set__` 方法使用托管实例中的同名属性覆盖（即插手接管）了要设置的属性。不过，也有非覆盖型描述符。下一节会详述这两种描述符之间的区别。

20.2　覆盖型与非覆盖型描述符对比

如前所述，Python 存取属性的方式特别不对等。通过实例读取属性时，通常返回的是实例中定义的属性；但是，如果实例中没有指定的属性，那么会获取类属性。而为实例中的属性赋值时，通常会在实例中创建属性，根本不影响类。

这种不对等的处理方式对描述符也有影响。其实，根据是否定义 `__set__` 方法，描述符可分为两大类。若想观察这两类描述符的行为差异，则需要使用几个类。我们将使用示例 20-8 中的代码作为接下来几节的试验台。

> 在示例 20-8 中，每个 `__get__` 和 `__set__` 方法都调用了 `print_args` 函数，使调用方式易于阅读。没必要深入理解 `print_args` 函数及辅助函数 `cls_name` 和 `display`，因此不要花心思研究它们。

示例 **20-8**　descriptorkinds.py：几个简单的类，用于研究描述符的覆盖行为

```
### 辅助函数，仅用于显示 ###

def cls_name(obj_or_cls):
    cls = type(obj_or_cls)
    if cls is type:
        cls = obj_or_cls
    return cls.__name__.split('.')[-1]

def display(obj):
    cls = type(obj)
    if cls is type:
        return '<class {}>'.format(obj.__name__)
    elif cls in [type(None), int]:
        return repr(obj)
    else:
        return '<{} object>'.format(cls_name(obj))
```

```
def print_args(name, *args):
    pseudo_args = ', '.join(display(x) for x in args)
    print('-> {}.__{}__({})'.format(cls_name(args[0]), name, pseudo_args))


### 对这个示例重要的类 ###

class Overriding:  ❶
    """也称数据描述符或强制描述符"""

    def __get__(self, instance, owner):
        print_args('get', self, instance, owner)  ❷

    def __set__(self, instance, value):
        print_args('set', self, instance, value)


class OverridingNoGet:  ❸
    """没有``__get__``方法的覆盖型描述符"""

    def __set__(self, instance, value):
        print_args('set', self, instance, value)


class NonOverriding:  ❹
    """也称非数据描述符或遮盖型描述符"""

    def __get__(self, instance, owner):
        print_args('get', self, instance, owner)


class Managed:  ❺
    over = Overriding()
    over_no_get = OverridingNoGet()
    non_over = NonOverriding()

    def spam(self):  ❻
        print('-> Managed.spam({})'.format(display(self)))
```

❶ 有 `__get__` 和 `__set__` 方法的典型覆盖型描述符。

❷ 在这个示例中，各个描述符的每个方法都调用了 `print_args` 函数。

❸ 没有 `__get__` 方法的覆盖型描述符。

❹ 没有 `__set__` 方法，所以这是非覆盖型描述符。

❺ 托管类，使用各个描述符类的一个实例。

❻ `spam` 方法放在这里是为了对比，因为方法也是描述符。

在接下来的几节中，我们要分析对 `Managed` 类及其实例做属性读写时的行为，还会讨论所定义的各个描述符。

20.2.1　覆盖型描述符

实现 `__set__` 方法的描述符属于覆盖型描述符，因为虽然描述符是类属性，但是实现 `__set__` 方法的话，会覆盖对实例属性的赋值操作。示例 20-2 就是这样实现的。特性也是覆盖型描述符：如果没提供设值函数，`property` 类中的 `__set__` 方法会抛出 `AttributeError` 异常，指明那个属性是只读的。我们可以使用示例 20-8 中的代码测试覆盖型描述符的行为，如示例 20-9 所示。

示例 **20-9**　覆盖型描述符的行为，其中 `obj.over` 是 `Overriding` 类（见示例 20-8）的实例

```
>>> obj = Managed()  ❶
>>> obj.over  ❷
-> Overriding.__get__(<Overriding object>, <Managed object>,
    <class Managed>)
>>> Managed.over  ❸
-> Overriding.__get__(<Overriding object>, None, <class Managed>)
>>> obj.over = 7  ❹
-> Overriding.__set__(<Overriding object>, <Managed object>, 7)
>>> obj.over  ❺
-> Overriding.__get__(<Overriding object>, <Managed object>,
    <class Managed>)
>>> obj.__dict__['over'] = 8  ❻
>>> vars(obj)  ❼
{'over': 8}
>>> obj.over  ❽
-> Overriding.__get__(<Overriding object>, <Managed object>,
    <class Managed>)
```

❶ 创建供测试使用的 Managed 对象。

❷ obj.over 触发描述符的 __get__ 方法，第二个参数的值是托管实例 obj。

❸ Managed.over 触发描述符的 __get__ 方法，第二个参数（instance）的值是 None。

❹ 为 obj.over 赋值，触发描述符的 __set__ 方法，最后一个参数的值是 7。

❺ 读取 obj.over，仍会触发描述符的 __get__ 方法。

❻ 跳过描述符，直接通过 obj.__dict__ 属性设值。

❼ 确认值在 obj.__dict__ 属性中，在 over 键名下。

❽ 然而，即使是名为 over 的实例属性，Managed.over 描述符仍会覆盖读取 obj.over 这个操作。

20.2.2　没有 __get__ 方法的覆盖型描述符

通常，覆盖型描述符既会实现 __set__ 方法，也会实现 __get__ 方法，不过也可以只实现 __set__ 方法，如示例 20-1 所示。此时，只有写操作由描述符处理。通过实例读取描述符会返回描述符对象本身，因为没有处理读操作的 __get__ 方法。如果直接通过实例的 __dict__ 属性创建同名实例属性，以后再设置那个属性时，仍会由 __set__ 方法插手接管，但是读取那个属性的话，就会直接从实例中返回新赋予的值，而不会返回描述符对象。也就是说，实例属性会遮盖描述符，不过只有读操作是如此。参见示例 20-10。

示例 **20-10**　没有 __get__ 方法的覆盖型描述符，其中 obj.over_no_get 是 OverridingNoGet 类（见示例 20-8）的实例

```
>>> obj.over_no_get  ❶
<__main__.OverridingNoGet object at 0x665bcc>
>>> Managed.over_no_get  ❷
```

```
<__main__.OverridingNoGet object at 0x665bcc>
>>> obj.over_no_get = 7  ❸
-> OverridingNoGet.__set__(<OverridingNoGet object>, <Managed object>, 7)
>>> obj.over_no_get  ❹
<__main__.OverridingNoGet object at 0x665bcc>
>>> obj.__dict__['over_no_get'] = 9  ❺
>>> obj.over_no_get  ❻
9
>>> obj.over_no_get = 7  ❼
-> OverridingNoGet.__set__(<OverridingNoGet object>, <Managed object>, 7)
>>> obj.over_no_get  ❽
9
```

❶ 这个覆盖型描述符没有 `__get__` 方法，因此，`obj.over_no_get` 从类中获取描述符实例。

❷ 直接从托管类中读取描述符实例也是如此。

❸ 为 `obj.over_no_get` 赋值会触发描述符的 `__set__` 方法。

❹ 因为 `__set__` 方法没有修改属性，所以在此读取 `obj.over_no_get` 获取的仍是托管类中的描述符实例。

❺ 通过实例的 `__dict__` 属性设置名为 `over_no_get` 的实例属性。

❻ 现在，`over_no_get` 实例属性会遮盖描述符，但是只有读操作是如此。

❼ 为 `obj.over_no_get` 赋值，仍然经过描述符的 `__set__` 方法处理。

❽ 但是读取时，只要有同名的实例属性，描述符就会被遮盖。

20.2.3　非覆盖型描述符

没有实现 `__set__` 方法的描述符是非覆盖型描述符。如果设置了同名的实例属性，描述符会被遮盖，致使描述符无法处理那个实例的那个属性。方法是以非覆盖型描述符实现的。示例 20-11 展示了对一个非覆盖型描述符的操作。

示例 **20-11**　非覆盖型描述符的行为，其中 `obj.non_over` 是 `NonOverriding` 类（见示例 20-8）的实例

```
>>> obj = Managed()
>>> obj.non_over  ❶
-> NonOverriding.__get__(<NonOverriding object>, <Managed object>,
     <class Managed>)
>>> obj.non_over = 7  ❷
>>> obj.non_over  ❸
7
>>> Managed.non_over  ❹
-> NonOverriding.__get__(<NonOverriding object>, None, <class Managed>)
>>> del obj.non_over  ❺
>>> obj.non_over  ❻
-> NonOverriding.__get__(<NonOverriding object>, <Managed object>,
     <class Managed>)
```

❶ `obj.non_over` 触发描述符的 `__get__` 方法，第二个参数的值是 `obj`。

❷ `Managed.non_over` 是非覆盖型描述符，因此没有干涉赋值操作的 `__set__` 方法。

❸ 现在，`obj` 有个名为 `non_over` 的实例属性，把 `Managed` 类的同名描述符属性遮盖掉。

❹ `Managed.non_over` 描述符依然存在，会通过类截获这次访问。

❺ 如果把 `non_over` 实例属性删除了……

❻ 那么，读取 `obj.non_over` 时，会触发类中描述符的 `__get__` 方法；但要注意，第二个参数的值是托管实例。

Python 贡献者和作者讨论这些概念时会使用不同的术语。覆盖型描述符也叫数据描述符或强制描述符。非覆盖型描述符也叫非数据描述符或遮盖型描述符。

在上述几个示例中，我们为几个与描述符同名的实例属性赋了值，结果依描述符中是否有 `__set__` 方法而有所不同。

依附在类上的描述符无法控制为类属性赋值的操作。其实，这意味着为类属性赋值能覆盖描述符属性，正如下一节所述的。

20.2.4　在类中覆盖描述符

不管描述符是不是覆盖型，为类属性赋值都能覆盖描述符。这是一种猴子补丁技术，不过在示例 20-12 中，我们把描述符替换成了整数，这其实会导致依赖描述符的类不能正确地执行操作。

示例 **20-12**　通过类可以覆盖任何描述符

```
>>> obj = Managed()  ❶
>>> Managed.over = 1  ❷
>>> Managed.over_no_get = 2
>>> Managed.non_over = 3
>>> obj.over, obj.over_no_get, obj.non_over  ❸
(1, 2, 3)
```

❶ 为后面的测试新建一个实例。

❷ 覆盖类中的描述符属性。

❸ 描述符真的不见了。

示例 20-12 揭示了读写属性的另一种不对等：读类属性的操作可以由依附在托管类上定义有 `__get__` 方法的描述符处理，但是写类属性的操作不会由依附在托管类上定义有 `__set__` 方法的描述符处理。

> 若想控制设置类属性的操作，要把描述符依附在类的类上，即依附在元类上。默认情况下，对用户定义的类来说，其元类是 `type`，而我们不能为 `type` 添加属性。不过在第 21 章，我们会自己创建元类。

下面我们调转话题，分析 Python 是如何使用描述符实现方法的。

20.3　方法是描述符

在类中定义的函数属于绑定方法（bound method），因为用户定义的函数都有 __get__ 方法，所以依附到类上时，就相当于描述符。示例 20-13 演示了从示例 20-8 里定义的 Managed 类中读取 spam 方法。

示例 **20-13**　方法是非覆盖型描述符

```
>>> obj = Managed()
>>> obj.spam  ❶
<bound method Managed.spam of <descriptorkinds.Managed object at 0x74c80c
>>> Managed.spam  ❷
<function Managed.spam at 0x734734>
>>> obj.spam = 7  ❸
>>> obj.spam
7
```

❶ obj.spam 获取的是绑定方法对象。

❷ 但是 Managed.spam 获取的是函数。

❸ 如果为 obj.spam 赋值，会遮盖类属性，导致无法通过 obj 实例访问 spam 方法。

函数没有实现 __set__ 方法，因此是非覆盖型描述符，如示例 20-13 中的最后一行所示。

从示例 20-13 中还可以看出一个重要信息：obj.spam 和 Managed.spam 获取的是不同的对象。与描述符一样，通过托管类访问时，函数的 __get__ 方法会返回自身的引用。但是，通过实例访问时，函数的 __get__ 方法返回的是绑定方法对象：一种可调用的对象，里面包装着函数，并把托管实例（例如 obj）绑定给函数的第一个参数（即 self），这与 functools.partial 函数的行为一致（参见 5.10.2 节）。

为了深入理解这种机制，请看示例 20-14。

示例 **20-14**　method_is_descriptor.py：`Text` 类，继承自 `UserString` 类

```
import collections


class Text(collections.UserString):

    def __repr__(self):
        return 'Text({!r})'.format(self.data)

    def reverse(self):
        return self[::-1]
```

下面来分析 `Text.reverse` 方法，如示例 20-15 所示。

示例 **20-15**　测试一个方法

```
    >>> word = Text('forward')
    >>> word  ❶
    Text('forward')
    >>> word.reverse()  ❷
    Text('drawrof')
    >>> Text.reverse(Text('backward'))  ❸
    Text('drawkcab')
    >>> type(Text.reverse), type(word.reverse)  ❹
    (<class 'function'>, <class 'method'>)
    >>> list(map(Text.reverse, ['repaid', (10, 20, 30), Text('stressed')]))
    ['diaper', (30, 20, 10), Text('desserts')]
    >>> Text.reverse.__get__(word)  ❻
    <bound method Text.reverse of Text('forward')>
    >>> Text.reverse.__get__(None, Text)  ❼
    <function Text.reverse at 0x101244e18>
    >>> word.reverse  ❽
    <bound method Text.reverse of Text('forward')>
    >>> word.reverse.__self__  ❾
    Text('forward')
    >>> word.reverse.__func__ is Text.reverse  ❿
    True
```

❶ `Text` 实例的 `repr` 方法返回一个类似 `Text` 构造方法调用的字符串，可用于创建相同的实例。

❷ reverse 方法返回反向拼写的单词。

❸ 在类上调用方法相当于调用函数。

❹ 注意类型是不同的，一个是 function，一个是 method。

❺ Text.reverse 相当于函数，甚至可以处理 Text 实例之外的其他对象。

❻ 函数都是非覆盖型描述符。在函数上调用 __get__ 方法时传入实例，得到的是绑定到那个实例上的方法。

❼ 调用函数的 __get__ 方法时，如果 instance 参数的值是 None，那么得到的是函数本身。

❽ word.reverse 表达式其实会调用 Text.reverse.__get__(word)，返回对应的绑定方法。

❾ 绑定方法对象有个 __self__ 属性，其值是调用这个方法的实例引用。

❿ 绑定方法的 __func__ 属性是依附在托管类上那个原始函数的引用。

绑定方法对象还有个 __call__ 方法，用于处理真正的调用过程。这个方法会调用 __func__ 属性引用的原始函数，把函数的第一个参数设为绑定方法的 __self__ 属性。这就是形参 self 的隐式绑定方式。

函数会变成绑定方法，这是 Python 语言底层使用描述符的最好例证。

深入了解描述符和方法的运作方式之后，下面讨论用法方面的一些实用建议。

20.4　描述符用法建议

下面根据刚刚论述的描述符特征给出一些实用的结论。

使用特性以保持简单

内置的 property 类创建的其实是覆盖型描述符，__set__ 方法和 __get__ 方法都实现了，即便不定义设值方法也是如此。特性的 __set__ 方法默认抛出 AttributeError: can't set attribute，因此创建只读属性最简单的方式是使用特性，这能避免下一条所述的问题。

只读描述符必须有 **__set__** 方法

如果使用描述符类实现只读属性，要记住，__get__ 和 __set__ 两个方法必须都定义，否则，实例的同名属性会遮盖描述符。只读属性的 __set__ 方法只需抛出 AttributeError 异常，并提供合适的错误消息。[6]

> [6]Python 为此类异常提供的错误消息不一致。如果试图修改 complex 的 c.real 属性，那么得到的错误消息是 AttributeError: read-only attribute；但是，如果试图修改 c.conjugat（e complex 对象的方法），那么得到的错误消息是 AttributeError: 'complex' object attribute 'conjugate' is read-only。

用于验证的描述符可以只有 **__set__** 方法

对仅用于验证的描述符来说，__set__ 方法应该检查 value 参数获得的值，如果有效，使用描述符实例的名称为键，直接在实例的 __dict__ 属性中设置。这样，从实例中读取同名属性的速度很快，因为不用经过 __get__ 方法处理。参见示例 20-1 中的代码。

仅有 **__get__** 方法的描述符可以实现高效缓存

如果只编写了 __get__ 方法，那么创建的是非覆盖型描述符。这种描述符可用于执行某些耗费资源的计算，然后为实例设置同名属性，缓存结果。同名实例属性会遮盖描述符，因此后续访问会直接从实例的

__dict__ 属性中获取值，而不会再触发描述符的 __get__ 方法。

非特殊的方法可以被实例属性遮盖

由于函数和方法只实现了 __get__ 方法，它们不会处理同名实例属性的赋值操作。因此，像 my_obj.the_method = 7 这样简单赋值之后，后续通过该实例访问 the_method 得到的是数字 7——但是不影响类或其他实例。然而，特殊方法不受这个问题的影响。解释器只会在类中寻找特殊的方法，也就是说，repr(x) 执行的其实是 x.__class__.__repr__(x)，因此 x 的 __repr__ 属性对 repr(x) 方法调用没有影响。出于同样的原因，实例的 __getattr__ 属性不会破坏常规的属性访问规则。

实例的非特殊方法可以被轻松地覆盖，这听起来不可靠且容易出错，可是在我使用 Python 的 15 年中从未受此困扰。然而，如果要创建大量动态属性，属性名称从不受自己控制的数据中获取（像本章前面那样），那么你应该知道这种行为；或许你还可以实现某种机制，过滤或转义动态属性的名称，以维持数据的健全性。

示例 19-6 中的 FrozenJSON 类不会出现实例属性遮盖方法的问题，因为那个类只有几个特殊方法和一个 build 类方法。只要通过类访问，类方法就是安全的，在示例 19-6 中我就是这么调用 FrozenJSON.build 方法的——在示例 19-7 中替换成 __new__ 方法了。Record 类（见示例 19-9 和示例 19-11）及其子类也是安全的，因为只用到了特殊的方法、类方法、静态方法和特性。特性是数据描述符，因此不能被实例属性覆盖。

讨论特性时讲了两个功能，这里讨论的描述符还未涉及，结束本章之前我们来讲讲：文档和对删除托管属性的处理。

20.5　描述符的文档字符串和覆盖删除操作

描述符类的文档字符串用于注解托管类中的各个描述符实例。图 20-6 中的截图是 `LineItem` 类（见示例 20-7）及 `Quantity` 和 `NonBlank` 描述符（见示例 20-6）的帮助界面。

提供的信息有点不足。对 `LineItem` 类来说，如果能说明 `weight` 必须以千克为单位就好了。这对特性来说是小菜一碟，因为各个特性只处理特定的托管属性。可是对描述符来说， `weight` 和 `price` 使用的都是 `Quantity` 描述符类。[7]

> [7]定制各个描述符实例的帮助文本特别难。有一种方法是为各个描述符实例动态构建包装类。

讨论特性时还讲了一个细节，而这里讨论的描述符没有涉及，那就是对删除托管属性的处理。在描述符类中，实现常规的 `__get__` 和（或）`__set__` 方法之外，可以实现 `__delete__` 方法，或者只实现 `__delete__` 方法做到这一点。时间充足的读者可以编写一个没有实际作用的描述符类实现 `__delete__` 方法，就当作练习。

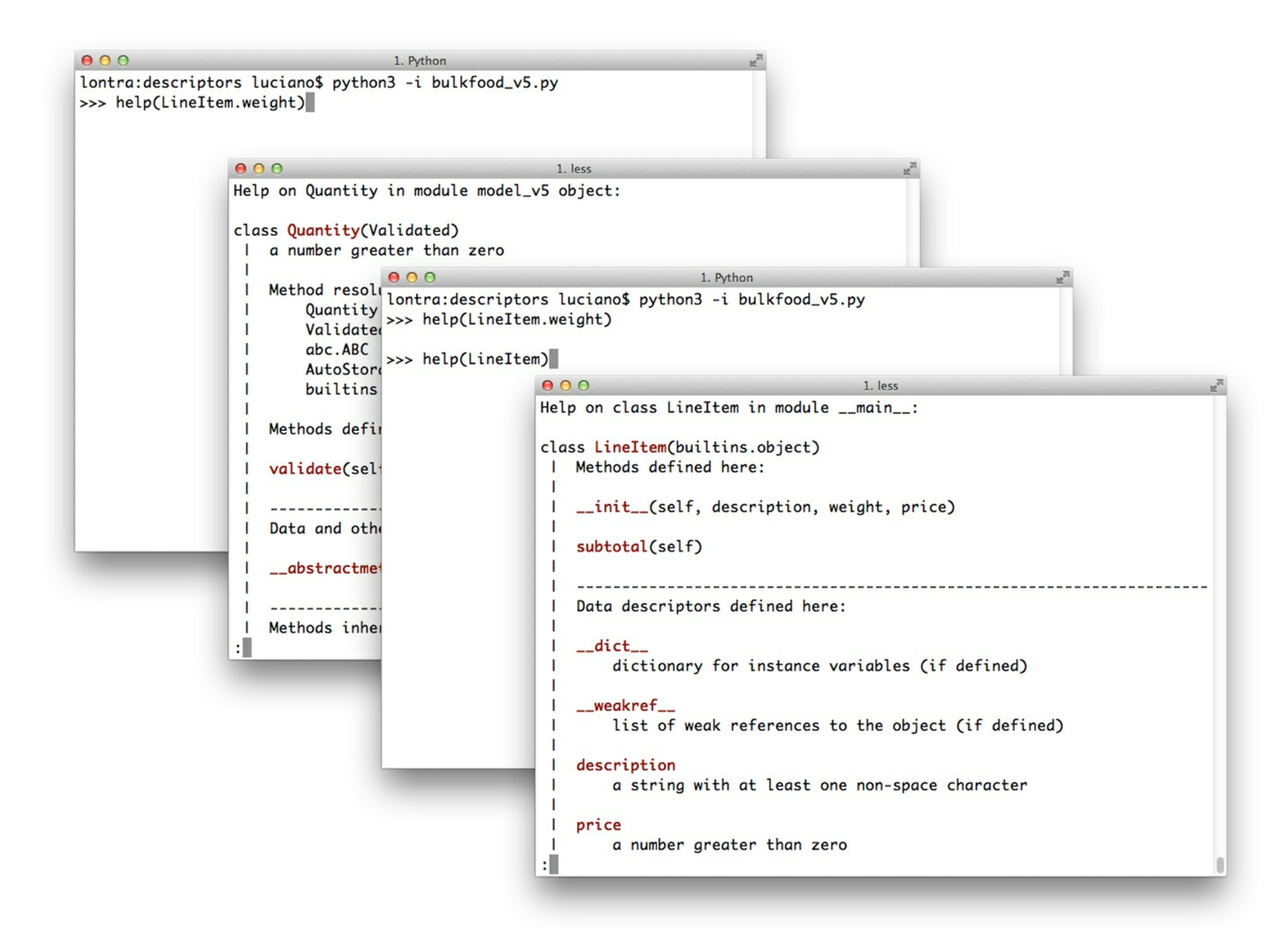


图 20-6：在 Python 控制台中执行 **`help(LineItem.weight)`** 和 **`help(LineItem)`** 命令时的截图

20.6 本章小结

本章的第一个示例接续第 19 章的 `LineItem` 系列示例。在示例 20-1 中，我们把特性替换成了描述符。我们知道，描述符类的实例能用作托管类的属性。为了讨论这个机制，我们引入了几个特殊的术语，例如托管实例和储存属性。

在 20.1.2 节，我们把声明 `Quantity` 描述符所需的 `storage_name` 参数去掉了，那个参数多余且容易出错，因为实例化描述符时指定的名称始终与赋值语句左边的属性名一样。我们采用的方法是，结合描述符类的名称和类中的计数器，生成独一无二的 `storage_name`（例如 `'_Quantity#1'`）。

接下来，本章对比了描述符类与使用函数式编程方式构建的特性工厂函数，分析了二者的代码量和优缺点。有时后者更合适也更简单，但是前者更灵活，而且是标准方案。20.1.3 节利用了描述符类的关键优势：通过子类共享代码，构建具有部分相同功能的专用描述符。

然后，我们分析了有或没有 `__set__` 方法时，描述符的行为有什么不同，了解了覆盖型描述符和非覆盖型描述符之间的重要差异。通过详细的测试，我们揭示了描述符何时接管，以及何时被遮盖、被跳过或被覆盖。

本章随后分析了非覆盖型描述符的一种具体类型：方法。通过控制台中的测试可知，通过实例访问依附在类上的函数时，经由描述符协议的处理，就会变成方法。

最后，我们对描述符的用法给出了一些建议，还简要说明了如何删除描述符和添加文档。

这一章我们遇到了几个只有类元编程能解决的问题，这些问题留到第 21 章解决。

20.7　延伸阅读

除了语言参考手册中必读的“Data model”一章（https://docs.python.org/3/reference/datamodel.html），Raymond Hettinger 写的“Descriptor HowTo Guide”（https://docs.python.org/3/howto/descriptor.html）也值得一读——这是 Python 官方文档 HowTo 合集（https://docs.python.org/3/howto/）中的一篇。

对 Python 对象模型相关的话题来说，Alex Martelli 写的《Python 技术手册（第 2 版）》一书虽然有点过时，但仍然提供了权威且客观的论述：本章讨论的关键机制在 Python 2.2 中引入，远在那本书涵盖的 2.5 版之前。Martelli 还做了一次题为“Python's Object Model”的演讲，深入探讨了特性和描述符 [幻灯片（http://www.aleax.it/Python/nylug05_om.pdf），视频（https://www.youtube.com/watch?v=VOzvpHoYQoo）]，强烈推荐观看。

至于针对 Python 3 的实例，David Beazley 与 Brian K. Jones 的《Python Cookbook（第 3 版）中文版》一书中有很多说明描述符的诀窍，推荐阅读的有“6.12 读取嵌套型和大小可变的二进制结构”“8.10 让属性具有惰性求值的能力”“8.13 实现一种数据模型或类型系统”和“9.9 把装饰器定义成类”。最后一个诀窍解决了函数装饰器、描述符和方法之间相互作用的深层次问题，说明了如何使用有 `__call__` 方法的类实现函数装饰器；如果既想装饰方法又想装饰函数，还要实现 `__get__` 方法。

杂谈

`self` 的问题

“变糟更好”（“Worse is Better”）是Richard P. Gabriel 在“The Rise of Worse is Better”一文（http://dreamsongs.com/RiseOfWorseIsBetter.html）中提出的设计思想。这个思想的第一要义是“简单”；对此，Gabriel 说道：

> 设计方式必须简单，对实现和接口来说都应如此。简单的实现比简单的接口更重要。简单是设计过程中最重要的考虑因素。

我认为，Python 要求明确把方法的第一个参数声明为 self 是“变糟更好”思想的体现。这样，实现是简单了（甚至也优雅了），但却牺牲了用户接口：方法的签名——例如 def zfill(self, width):——在外观上与 pobox.zfill(8) 调用不匹配。

这种做法（以及使用 self 这个标识符）由 Modula-3 语言创造，但是与 Python 有差异：在 Modula-3 中，接口的声明与实现是分开的，而且在接口声明中会省略 self 参数，因此对用户来说，接口声明中的方法显示的参数数量与真正接受的参数数量完全一致。

在这方面，Python 有一项改进——错误消息。对于用户定义的单参数（除 self 之外）方法来说，如果用户调用 obj.meth()，Python 2.7 会抛出异常，显示 TypeError: meth() takes exactly 2 arguments (1 given)；不过在 Python 3.4 中，错误消息没那么难以理解了，解决了参数数量问题，还指出了缺失的参数：meth() missing 1 required positional argument: 'x'。

除了要明确把 self 作为参数之外，限制必须使用 self 访问实例属性也备受批评。[8] 我自己并不介意输入 self 限定符，这样便于把局部变量和属性区分开。我介意的是在 def 语句中使用 self。但是我已经习惯了。

如果讨厌 Python 要求显式使用 self，可以想想 JavaScript 中隐式的 this 那变幻莫测的语义，这样感觉就会好多了。像这样使用self 有一些合理之处，Guido 在他的博客 The History of Python 中写了一篇文章，题为“Adding Support for User-defined Classes”（http://python-history.blogspot.com.br/2009/02/adding-support-for-user-defined-classes.html），说明了这些原因。

[8]例如，A. M. Kuchling 发表的著名文章“Python Warts”（存档：http://web.archive.org/web/20031002184114/www.amk.ca/python/writing/warts.html）。Kuchling 自己并不讨厌 self 限定符，但是他提到了这一点——可能是为了呼应 comp.lang.python 邮件列表中的观点。

第 21 章　类元编程

> （元类）是深奥的知识，99% 的用户都无需关注。如果你想知道是否需要使用元类，我告诉你，不需要（真正需要使用元类的人确信他们需要，无需解释原因）。[1]
>
> ——Tim Peters
> Timsort 算法的发明者，活跃的 Python 贡献者

[1]摘自 comp.lang.python 邮件列表中对“Acrimony in c.l.p.”话题的回复（https://mail.python.org/pipermail/python-list/2002-December/134521.html）。前言中引述的那句话也是出自这篇发布于 2002 年 12 月 23 日的消息。TimBot 在那天获得了灵感。

类元编程是指在运行时创建或定制类的技艺。在 Python 中，类是一等对象，因此任何时候都可以使用函数新建类，而无需使用 `class` 关键字。类装饰器也是函数，不过能够审查、修改，甚至把被装饰的类替换成其他类。最后，元类是类元编程最高级的工具：使用元类可以创建具有某种特质的全新类种，例如我们见过的抽象基类。

元类功能强大，但是难以掌握。类装饰器能使用更简单的方式解决很多问题。其实， Python 2.6 引入类装饰器之后，元类很难使用真实的代码说明，因此我不会像前面的章节那样再举引导示例。

本章还会谈及导入时和运行时的区别——这是有效使用 Python 元编程的重要基础。



> 这是一个令人兴奋的话题，很容易让人忘乎所以。因此，进入本章的正文之前，我必须告诫你：
>
> 除非开发框架，否则不要编写元类——然而，为了寻找乐趣，或者练习相关的概念，可以这么做。

首先，本章探讨如何在运行时创建类。

21.1　类工厂函数

本书多次提到标准库中的一个类工厂函数——`collections.namedtuple`。我们把一个类名和几个属性名传给这个函数，它会创建一个 `tuple` 的子类，其中的元素通过名称获取，还为调试提供了友好的字符串表示形式（`__repr__`）。

有时，我觉得应该有类似的工厂函数，用于创建可变对象。假设我在编写一个宠物店应用程序，我想把狗的数据当作简单的记录处理。编写下面的样板代码让人厌烦：

```
class Dog:
    def __init__(self, name, weight, owner):
        self.name = name
        self.weight = weight
        self.owner = owner
```

无趣……各个字段名称出现了三次。写了这么多样板代码，甚至字符串表示形式都不友好：

```
>>> rex = Dog('Rex', 30, 'Bob')
>>> rex
<__main__.Dog object at 0x2865bac>
```

参考 `collections.namedtuple`，下面我们创建一个 `record_factory` 函数，即时创建简单的类（如 `Dog`）。这个函数的用法如示例 21-1。

示例 **21-1**　测试 `record_factory` 函数，一个简单的类工厂函数

```
>>> Dog = record_factory('Dog', 'name weight owner')  ❶
>>> rex = Dog('Rex', 30, 'Bob')
>>> rex  ❷
Dog(name='Rex', weight=30, owner='Bob')
>>> name, weight, _ = rex  ❸
>>> name, weight
('Rex', 30)
```

```
>>> "{2}'s dog weighs {1}kg".format(*rex)  ❹
"Bob's dog weighs 30kg"
>>> rex.weight = 32  ❺
>>> rex
Dog(name='Rex', weight=32, owner='Bob')
>>> Dog.__mro__  ❻
(<class 'factories.Dog'>, <class 'object'>)
```

❶ 这个工厂函数的签名与 `namedtuple` 类似：先写类名，后面跟着写在一个字符串里的多个属性名，使用空格或逗号分开。

❷ 友好的字符串表示形式。

❸ 实例是可迭代的对象，因此赋值时可以便利地拆包。

❹ 传给 `format` 等函数时也可以拆包。

❺ 记录实例是可变的对象。

❻ 新建的类继承自 `object`，与我们的工厂函数没有关系。

`record_factory` 函数的代码在示例 21-2 中。[2]

> [2]感谢我的朋友 J.S. Bueno 的建议。

示例 **21-2** record_factory.py：一个简单的类工厂函数

```
def record_factory(cls_name, field_names):
    try:
        field_names = field_names.replace(',', ' ').split()  ❶
    except AttributeError:  # 不能调用.replace或.split方法
        pass  # 假定field_names本就是标识符组成的序列
    field_names = tuple(field_names)  ❷

    def __init__(self, *args, **kwargs):  ❸
        attrs = dict(zip(self.__slots__, args))
        attrs.update(kwargs)
        for name, value in attrs.items():
            setattr(self, name, value)

    def __iter__(self):  ❹
        for name in self.__slots__:
```

```
            yield getattr(self, name)

    def __repr__(self):  ❺
        values = ', '.join('{}={!r}'.format(*i) for i
                           in zip(self.__slots__, self))
        return '{}({})'.format(self.__class__.__name__, values)

    cls_attrs = dict(__slots__ = field_names,  ❻
                     __init__  = __init__,
                     __iter__  = __iter__,
                     __repr__  = __repr__)

    return type(cls_name, (object,), cls_attrs)  ❼
```

❶ 这里体现了鸭子类型：尝试在逗号或空格处拆分 field_names；如果失败，那么假定 field_names 本就是可迭代的对象，一个元素对应一个属性名。

❷ 使用属性名构建元组，这将成为新建类的 __slots__ 属性；此外，这么做还设定了拆包和字符串表示形式中各字段的顺序。

❸ 这个函数将成为新建类的 __init__ 方法。参数有位置参数和（或）关键字参数。

❹ 实现 __iter__ 函数，把类的实例变成可迭代的对象；按照 __slots__ 设定的顺序产出字段值。

❺ 迭代 __slots__ 和 self，生成友好的字符串表示形式。

❻ 组建类属性字典。

❼ 调用 type 构造方法，构建新类，然后将其返回。

通常，我们把 type 视作函数，因为我们像函数那样使用它，例如，调用 type(my_object) 获取对象所属的类——作用与 my_object.__class__ 相同。然而，type 是一个类。当成类使用时，传入三个参数可以新建一个类：

```
MyClass = type('MyClass', (MySuperClass, MyMixin),
               {'x': 42, 'x2': lambda self: self.x * 2})
```

`type` 的三个参数分别是 `name`、`bases` 和 `dict`。最后一个参数是一个映射，指定新类的属性名和值。上述代码的作用与下述代码相同：

```
class MyClass(MySuperClass, MyMixin):
    x = 42

    def x2(self):
        return self.x * 2
```

让人觉得新奇的是，`type` 的实例是类，例如这里的 `MyClass` 类或示例 21-1 中的 `Dog` 类。

总之，示例 21-2 中 `record_factory` 函数的最后一行会构建一个类，类的名称是 `cls_name` 参数的值，唯一的直接超类是 `object`，有 `__slots__`、`__init__`、`__iter__` 和 `__repr__` 四个类属性，其中后三个是实例方法。

我们本可以把 `__slots__` 类属性的名称改成其他值，不过要是那样的话，就要实现 `__setattr__` 方法，为属性赋值时验证属性的名称，因为对于记录这样的类，我们希望属性始终是固定的那几个，而且顺序相同。然而 9.8 节说过，`__slots__` 属性的主要特色是节省内存，能处理数百万个实例，不过也有一些缺点。

把三个参数传给 `type` 是动态创建类的常用方式。如果查看 `collections.namedtuple` 函数的源码（https://hg.python.org/cpython/file/3.4/Lib/collections/__init__.py#l236），你会发现另一种方式：先声明一个 `_class_template` 变量，其值是字符串形式的源码模板；然后在 `namedtuple` 函数中调用 `_class_template.format(...)` 方法，填充模板里的空白；最后，使用内置的 `exec` 函数计算得到的源码字符串。

在 Python 中做元编程时，最好不用 `exec` 和 `eval` 函数。如果接收的字符串（或片段）来自不可信的源，那么这两个函数会带来严重的安全风险。Python 提供了充足的内省工具，大多数时候都

不需要使用 exec 和 eval 函数。然而，Python 核心开发者实现 namedtuple 函数时选择了使用 exec 函数，这样做是为了让生成的类代码能通过 ._source 属性（https://docs.python.org/3/library/collections.html#collections.somenamed
获取。

record_factory 函数创建的类，其实例有个局限——不能序列化，即不能使用 pickle 模块里的 dump/load 函数处理。这个示例是为了说明如何使用 type 类满足简单的需求，因此不会解决这个问题。如果想了解完整的方案，请分析 collections.nameduple 函数的源码（https://hg.python.org/cpython/file/3.4/Lib/collections/__init__.py#l236），搜索“pickling”这个词。

21.2　定制描述符的类装饰器

20.1.3 节中的 `LineItem` 示例还有个问题没有解决：储存属性的名称不具有描述性，即属性（如 `weight`）的值存储在名为 `_Quantity#0` 的实例属性中，这样的名称有点不便于调试。我们可以使用下述代码从示例 20-7 定义的描述符中获取储存属性的名称：

```
>>> LineItem.weight.storage_name
'_Quantity#0'
```

可是，如果储存属性的名称中包含托管属性的名称更好，如下所示：

```
>>> LineItem.weight.storage_name
'_Quantity#weight'
```

20.1.2 节说过，我们不能使用描述性的储存属性名称，因为实例化描述符时无法得知托管属性（即绑定到描述符上的类属性，例如前述示例中的 `weight`）的名称。可是，一旦组建好整个类，而且把描述符绑定到类属性上之后，我们就可以审查类，并为描述符设置合理的储存属性名称。`LineItem` 类的 `__new__` 方法可以做到这一点，因此，在 `__init__` 方法中使用描述符时，储存属性已经设置了正确的名称。为了解决这个问题而使用 `__new__` 方法纯属白费力气：每次新建 `LineItem` 实例时都会运行 `__new__` 方法中的逻辑，可是，一旦 `LineItem` 类构建好了，描述符与托管属性之间的绑定就不会变了。因此，我们要在创建类时设置储存属性的名称。使用类装饰器或元类可以做到这一点。我们首先使用较简单的方式。

类装饰器与函数装饰器非常类似，是参数为类对象的函数，返回原来的类或修改后的类。

在示例 21-3 中，解释器会计算 `LineItem` 类，把返回的类对象传给 `model.entity` 函数。Python 会把 `LineItem` 这个全局名称绑定给 `model.entity` 函数返回的对象。在这个示例中，`model.entity` 函数会返回原先的 `LineItem` 类，但是会修改各个描述符实例的 `storage_name` 属性。

示例 **21-3**　bulkfood_v6.py：使用 `Quantity` 和 `NonBlank` 描述符的 `LineItem` 类

```
import model_v6 as model

@model.entity  ❶
class LineItem:
    description = model.NonBlank()
    weight = model.Quantity()
    price = model.Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ 这个类唯一的变化是添加了装饰器。

示例 21-4 是那个装饰器的实现。这里只列出了 model_v6.py 脚本底部添加的新代码，其余的代码与 model_v5.py 脚本（见示例 20-6）一样。

示例 **21-4**　model_v6.py：一个类装饰器

```
def entity(cls):  ❶
    for key, attr in cls.__dict__.items():  ❷
        if isinstance(attr, Validated):  ❸
            type_name = type(attr).__name__
            attr.storage_name = '_{}#{}'.format(type_name, key)  ❹
    return cls  ❺
```

❶ 装饰器的参数是一个类。

❷ 迭代存储类属性的字典。

❸ 如果属性是 `Validated` 描述符的实例……

❹ ……使用描述符类的名称和托管属性的名称命名 `storage_name`（例

如 _NonBlank#description）。

❺ 返回修改后的类。

bulkfood_v6.py 脚本中的 doctest 证明，改动是成功的。例如，示例 21-5 展示了一个 LineItem 实例中的储存属性名称。

示例 **21-5**　bulkfood_v6.py：描述符中新 storage_name 属性的 doctest

```
>>> raisins = LineItem('Golden raisins', 10, 6.95)
>>> dir(raisins)[:3]
['_NonBlank#description', '_Quantity#price', '_Quantity#weight']
>>> LineItem.description.storage_name
'_NonBlank#description'
>>> raisins.description
'Golden raisins'
>>> getattr(raisins, '_NonBlank#description')
'Golden raisins'
```

可以看出，这并不复杂。类装饰器能以较简单的方式做到以前需要使用元类去做的事情——创建类时定制类。

类装饰器有个重大缺点：只对直接依附的类有效。这意味着，被装饰的类的子类可能继承也可能不继承装饰器所做的改动，具体情况视改动的方式而定。接下来的几节会探讨这个问题，并给出解决方案。

21.3　导入时和运行时比较

为了正确地做元编程，你必须知道 Python 解释器什么时候计算各个代码块。Python 程序员会区分“导入时”和“运行时”，不过这两个术语没有严格的定义，而且二者之间存在着灰色地带。在导入时，解释器会从上到下一次性解析完 .py 模块的源码，然后生成用于执行的字节码。如果句法有错误，就在此时报告。如果本地的 `__pycache__` 文件夹中有最新的 .pyc 文件，解释器会跳过上述步骤，因为已经有运行所需的字节码了。

编译肯定是导入时的活动，不过那个时期还会做些其他事，因为 Python 中的语句几乎都是可执行的，也就是说语句可能会运行用户代码，修改用户程序的状态。尤其是 `import` 语句，它不只是声明 [3]，在进程中首次导入模块时，还会运行所导入模块中的全部顶层代码——以后导入相同的模块则使用缓存，只做名称绑定。那些顶层代码可以做任何事，包括通常在“运行时”做的事，例如连接数据库。[4] 因此，“导入时”与“运行时”之间的界线是模糊的：`import` 语句可以触发任何“运行时”行为。

> [3]Java 中的 `import` 语句则只是声明，用于告知编译器需要特定的包。

> [4]我不是说导入模块时应该连接数据库，只是指出来可以做到。

在前一段中我写道，导入时会“运行全部顶层代码”，但是“顶层代码”会经过一些加工。导入模块时，解释器会执行顶层的 `def` 语句，可是这么做有什么作用呢？解释器会编译函数的定义体（首次导入模块时），把函数对象绑定到对应的全局名称上，但是显然解释器不会执行函数的定义体。通常这意味着解释器在导入时定义顶层函数，但是仅当在运行时调用函数时才会执行函数的定义体。

对类来说，情况就不同了：在导入时，解释器会执行每个类的定义体，甚至会执行嵌套类的定义体。执行类定义体的结果是，定义了类的属性和方法，并构建了类对象。从这个意义上理解，类的定义体属于“顶层代码”，因为它在导入时运行。

上述说明模糊又抽象，下面通过练习理解各个时期所做的事情。

理解计算时间的练习

假设在 evaltime.py 脚本中导入了 evalsupport.py 模块。这两个模块调用了几次 `print` 函数，打印 `<[N]>` 格式的标记，其中 `N` 是数字。下述两个练习的目标是，确定各个调用在何时执行。

> 据我的学生说，这两个练习有助于更好地理解 Python 计算源码的方式。在查看场景 1 的解答之前，请一定要拿出纸和笔，花点时间作答。

那两个模块的代码在示例 21-6 和示例 21-7 中。先别运行代码，拿出纸和笔，按顺序写出下述两个场景输出的标记。

场景 **1**

在 Python 控制台中以交互的方式导入 evaltime.py 模块：

```
>> import evaltime
```

场景 **2**

在命令行中运行 evaltime.py 模块：

```
$ python3 evaltime.py
```

示例 **21-6**　evaltime.py：按顺序写出输出的序号标记 `<[N]>`

```
from evalsupport import deco_alpha

print('<[1]> evaltime module start')


class ClassOne():
    print('<[2]> ClassOne body')

    def __init__(self):
        print('<[3]> ClassOne.__init__')

    def __del__(self):
```

```
        print('<[4]> ClassOne.__del__')

    def method_x(self):
        print('<[5]> ClassOne.method_x')

    class ClassTwo(object):
        print('<[6]> ClassTwo body')


@deco_alpha
class ClassThree():
    print('<[7]> ClassThree body')

    def method_y(self):
        print('<[8]> ClassThree.method_y')


class ClassFour(ClassThree):
    print('<[9]> ClassFour body')

    def method_y(self):
        print('<[10]> ClassFour.method_y')


if __name__ == '__main__':
    print('<[11]> ClassOne tests', 30 * '.')
    one = ClassOne()
    one.method_x()
    print('<[12]> ClassThree tests', 30 * '.')
    three = ClassThree()
    three.method_y()
    print('<[13]> ClassFour tests', 30 * '.')
    four = ClassFour()
    four.method_y()


print('<[14]> evaltime module end')
```

示例 **21-7**　evalsupport.py：evaltime.py 导入的模块

```
print('<[100]> evalsupport module start')

def deco_alpha(cls):
    print('<[200]> deco_alpha')
```

```
    def inner_1(self):
        print('<[300]> deco_alpha:inner_1')

    cls.method_y = inner_1
    return cls

class MetaAleph(type):
    print('<[400]> MetaAleph body')

    def __init__(cls, name, bases, dic):
        print('<[500]> MetaAleph.__init__')

        def inner_2(self):
            print('<[600]> MetaAleph.__init__:inner_2')

        cls.method_z = inner_2

print('<[700]> evalsupport module end')
```

01. 场景1的解答

在 Python 控制台中导入 evaltime.py 模块后得到的输出如示例 21-8 所示。

示例 **21-8**　场景 1：在 Python 控制台中导入 `evaltime` 模块

```
>>> import evaltime
<[100]> evalsupport module start ❶
<[400]> MetaAleph body ❷
<[700]> evalsupport module end
<[1]> evaltime module start
<[2]> ClassOne body ❸
<[6]> ClassTwo body ❹
<[7]> ClassThree body
<[200]> deco_alpha ❺
<[9]> ClassFour body
<[14]> evaltime module end ❻
```

❶ `evalsupport` 模块中的所有顶层代码在导入模块时运行；解释器会编译 `deco_alpha` 函数，但是不会执行定义体。

❷ `MetaAleph` 类的定义体运行了。

❸ 每个类的定义体都执行了……

❹ ……包括嵌套的类。

❺ 先计算被装饰的类 `ClassThree` 的定义体，然后运行装饰器函数。

❻ 在这个场景中，`evaltime` 模块是导入的，因此不会运行 `if __name__ == '__main__':` 块。

对于场景 1，要注意以下几点。

(1) 这个场景由简单的 `import evaltime` 语句触发。

(2) 解释器会执行所导入模块及其依赖（`evalsupport`）中的每个类定义体。

(3) 解释器先计算类的定义体，然后调用依附在类上的装饰器函数，这是合理的行为，因为必须先构建类对象，装饰器才有类对象可处理。

(4) 在这个场景中，只运行了一个用户定义的函数或方法——`deco_alpha` 装饰器。

下面来看场景 2。

02. 场景**2**的解答

运行 `python3 evaltime.py` 命令后得到的输出如示例 21-9 所示。

示例 **21-9**　场景 2：在 shell 中运行 evaltime.py

```
$ python3 evaltime.py
<[100]> evalsupport module start
<[400]> MetaAleph body
<[700]> evalsupport module end
<[1]> evaltime module start
<[2]> ClassOne body
<[6]> ClassTwo body
```

```
<[7]> ClassThree body
<[200]> deco_alpha
<[9]> ClassFour body ❶
<[11]> ClassOne tests ..............................
<[3]> ClassOne.__init__ ❷
<[5]> ClassOne.method_x
<[12]> ClassThree tests ..............................
<[300]> deco_alpha:inner_1 ❸
<[13]> ClassFour tests ..............................
<[10]> ClassFour.method_y
<[14]> evaltime module end
<[4]> ClassOne.__del__ ❹
```

❶ 目前为止，输出与示例 21-8 相同。

❷ 类的标准行为。

❸ `deco_alpha` 装饰器修改了 `ClassThree.method_y` 方法，因此调用 `three.method_y()` 时会运行 `inner_1` 函数的定义体。

❹ 只有程序结束时，绑定在全局变量 `one` 上的 `ClassOne` 实例才会被垃圾回收程序回收。

场景 2 主要想说明的是，类装饰器可能对子类没有影响。在示例 21-6 中，我们把 `ClassFour` 定义为 `ClassThree` 的子类。`ClassThree` 类上依附的 `@deco_alpha` 装饰器把 `method_y` 方法替换掉了，但是这对 `ClassFour` 类根本没有影响。当然，如果 `ClassFour.method_y` 方法使用 `super(...)` 调用 `ClassThree.method_y` 方法，我们便会看到装饰器起作用，执行 `inner_1` 函数。

与此不同的是，如果想定制整个类层次结构，而不是一次只定制一个类，使用下一节介绍的元类更高效。

21.4　元类基础知识

元类是制造类的工厂，不过不是函数（如示例 21-2 中的 `record_factory`），而是类。图 21-1 使用机器和小怪兽图示法描述元类，可以看出，元类是生产机器的机器。

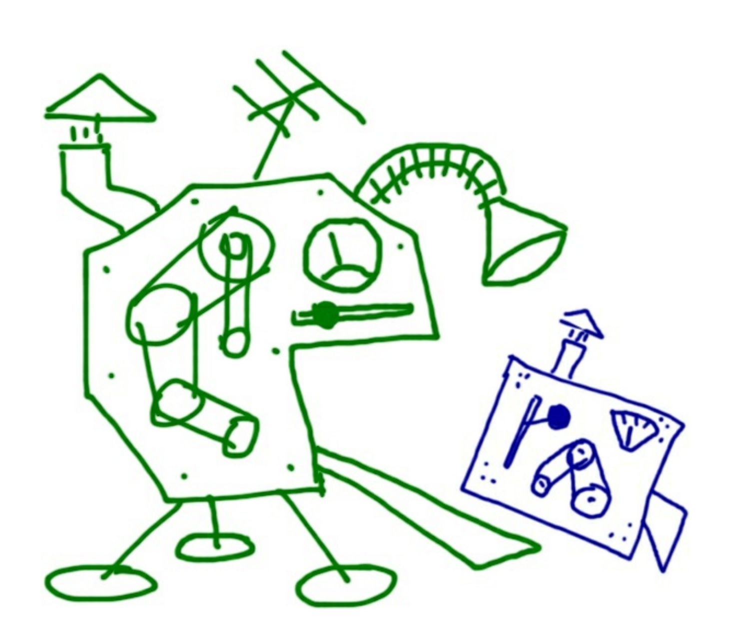




图 **21-1**：元类是用于构建类的类

根据 Python 对象模型，类是对象，因此类肯定是另外某个类的实例。默认情况下，Python 中的类是 `type` 类的实例。也就是说，`type` 是大多数内置的类和用户定义的类的元类：

```
>>> 'spam'.__class__
<class 'str'>
>>> str.__class__
<class 'type'>
>>> from bulkfood_v6 import LineItem
>>> LineItem.__class__
<class 'type'>
>>> type.__class__
<class 'type'>
```

为了避免无限回溯，`type` 是其自身的实例，如最后一行所示。

注意，我没有说 str 或 LineItem 继承自 type。我的意思是，str 和 LineItem 是 type 的实例。这两个类是 object 的子类。图 21-2 可能有助于你理清这个奇怪的现象。

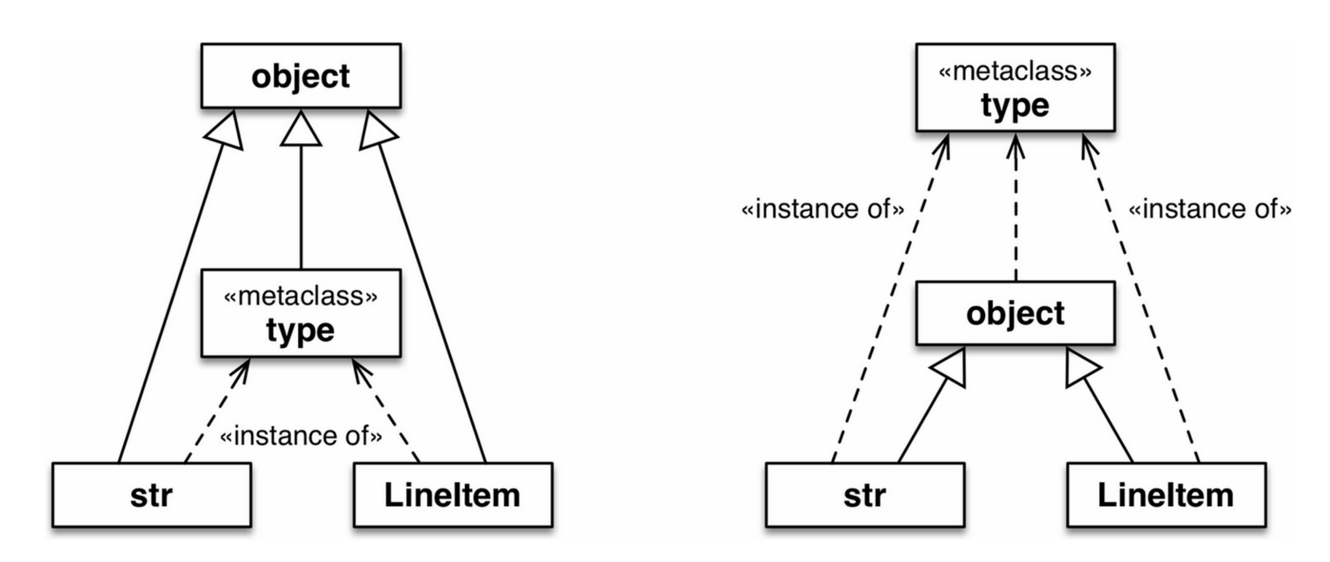


图 **21-2**：两个示意图都是正确的。左边的示意图强调 **str**、**type** 和 **LineItem** 是 **object** 的子类。右边的示意图则清楚地表明 **str**、**object** 和 **LineItem** 是 **type** 的实例，因为它们都是类

> object 类和 type 类之间的关系很独特：object 是 type 的实例，而 type 是 object 的子类。这种关系很“神奇”，无法使用 Python 代码表述，因为定义其中一个之前另一个必须存在。type 是自身的实例这一点也很神奇。

除了 type，标准库中还有一些别的元类，例如 ABCMeta 和 Enum。如下述代码片段所示，collections.Iterable 所属的类是 abc.ABCMeta。Iterable 是抽象类，而 ABCMeta 不是——不管怎样，Iterable 是 ABCMeta 的实例：

```
>>> import collections
>>> collections.Iterable.__class__
<class 'abc.ABCMeta'>
>>> import abc
>>> abc.ABCMeta.__class__
<class 'type'>
>>> abc.ABCMeta.__mro__
(<class 'abc.ABCMeta'>, <class 'type'>, <class 'object'>)
```

向上追溯，`ABCMeta` 最终所属的类也是 `type`。所有类都直接或间接地是 `type` 的实例，不过只有元类同时也是 `type` 的子类。若想理解元类，一定要知道这种关系：元类（如 `ABCMeta`）从 `type` 类继承了构建类的能力。图 21-3 对这种至关重要的关系做了图解。

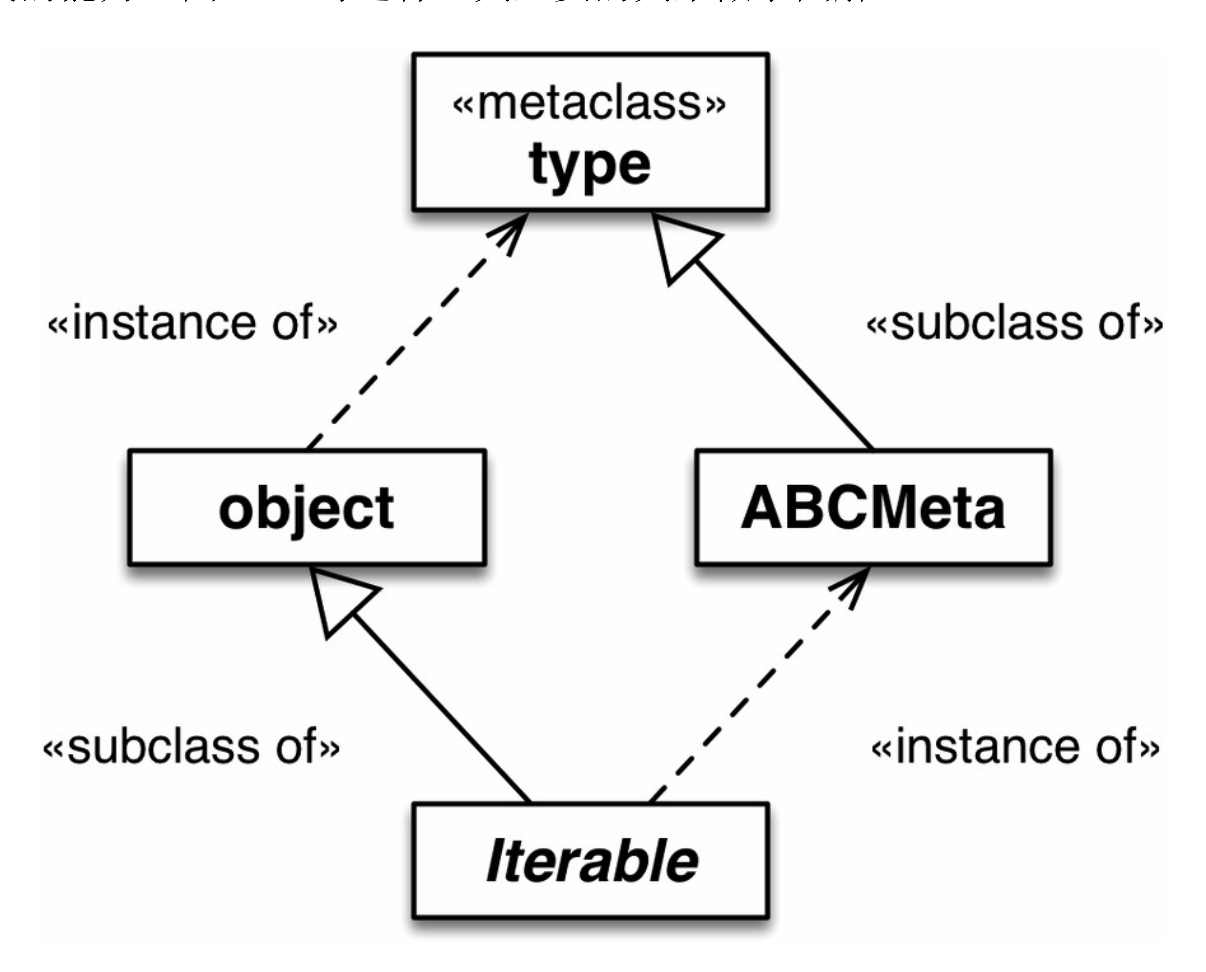


图 **21-3**：**`Iterable`** 是 **`object`** 的子类，是 **`ABCMeta`** 的实例。**`object`** 和 **`ABCMeta`** 都是 **`type`** 的实例，但是这里的重要关系是，**`ABCMeta`** 还是 **`type`** 的子类，因为 **`ABCMeta`** 是元类。示意图中只有 **`Iterable`** 是抽象类

我们要抓住的重点是，所有类都是 `type` 的实例，但是元类还是 `type` 的子类，因此可以作为制造类的工厂。具体来说，元类可以通过实现 `__init__` 方法定制实例。元类的 `__init__` 方法可以做到类装饰器能做的任何事情，但是作用更大，如接下来的练习所示。

理解元类计算时间的练习

我们对 21.3 节的练习做些改动，evalsupport.py 模块与示例 21-7 一样，不过现在主脚本变成 evaltime_meta.py 了，如示例 21-10 所示。

示例 **21-10**　evaltime_meta.py：`ClassFive` 是 `MetaAleph` 元类的实例

```
from evalsupport import deco_alpha
from evalsupport import MetaAleph

print('<[1]> evaltime_meta module start')


@deco_alpha
class ClassThree():
    print('<[2]> ClassThree body')

    def method_y(self):
        print('<[3]> ClassThree.method_y')


class ClassFour(ClassThree):
    print('<[4]> ClassFour body')

    def method_y(self):
        print('<[5]> ClassFour.method_y')


class ClassFive(metaclass=MetaAleph):
    print('<[6]> ClassFive body')

    def __init__(self):
        print('<[7]> ClassFive.__init__')

    def method_z(self):
        print('<[8]> ClassFive.method_z')


class ClassSix(ClassFive):
    print('<[9]> ClassSix body')

    def method_z(self):
        print('<[10]> ClassSix.method_z')


if __name__ == '__main__':
    print('<[11]> ClassThree tests', 30 * '.')
```

```
    three = ClassThree()
    three.method_y()
    print('<[12]> ClassFour tests', 30 * '.')
    four = ClassFour()
    four.method_y()
    print('<[13]> ClassFive tests', 30 * '.')
    five = ClassFive()
    five.method_z()
    print('<[14]> ClassSix tests', 30 * '.')
    six = ClassSix()
    six.method_z()

print('<[15]> evaltime_meta module end')
```

同样，请拿出纸和笔，按顺序写出下述两个场景中输出的序号标记`<[N]>`。

场景 **3**

在 Python 控制台中以交互的方式导入 evaltime_meta.py 模块。

场景 **4**

在命令行中运行 evaltime_meta.py 模块。

解答和分析如下。

01. 场景**3**的解答

在 Python 控制台中导入 evaltime_meta.py 模块后得到的输出如示例 21-11 所示。

示例 **21-11** 场景 3：在 Python 控制台中导入 `evaltime_meta` 模块

```
>>> import evaltime_meta
<[100]> evalsupport module start
<[400]> MetaAleph body
<[700]> evalsupport module end
<[1]> evaltime_meta module start
<[2]> ClassThree body
```

```
<[200]> deco_alpha
<[4]> ClassFour body
<[6]> ClassFive body
<[500]> MetaAleph.__init__ ❶
<[9]> ClassSix body
<[500]> MetaAleph.__init__ ❷
<[15]> evaltime_meta module end
```

❶ 与场景 1 的关键区别是，创建 `ClassFive` 时调用了 `MetaAleph.__init__` 方法。

❷ 创建 `ClassFive` 的子类 `ClassSix` 时也调用了 `MetaAleph.__init__` 方法。

Python 解释器计算 `ClassFive` 类的定义体时没有调用 `type` 构建具体的类定义体，而是调用 `MetaAleph` 类。看一下示例 21-12 中定义的 `MetaAleph` 类，你会发现 `__init__` 方法有四个参数。

`self`

　　这是要初始化的类对象（例如 `ClassFive`）。

`name`、`bases`、`dic`

　　与构建类时传给 `type` 的参数一样。

示例 **21-12**　evalsupport.py：定义 `MetaAleph` 元类，摘自示例 21-7

```
class MetaAleph(type):
    print('<[400]> MetaAleph body')

    def __init__(cls, name, bases, dic):
        print('<[500]> MetaAleph.__init__')

        def inner_2(self):
            print('<[600]> MetaAleph.__init__:inner_2')

        cls.method_z = inner_2
```

编写元类时，通常会把 `self` 参数改成 `cls`。例如，在上述元类的 `__init__` 方法中，把第一个参数命名为 `cls` 能清楚地表明要构建的实例是类。

`__init__` 方法的定义体中定义了 `inner_2` 函数，然后将其绑定给 `cls.method_z`。`MetaAleph.__init__` 方法签名中的 `cls` 指代要创建的类（例如 `ClassFive`）。而 `inner_2` 函数签名中的 `self` 最终是指代我们在创建的类的实例（例如 `ClassFive` 类的实例）。

02. 场景**4**的解答

在命令行中运行 `python3 evaltime_meta.py` 命令后得到的输出如示例 21-13 所示。

示例 **21-13**　场景 4：在 shell 中运行 evaltime_meta.py

```
$ python3 evaltime.py
<[100]> evalsupport module start
<[400]> MetaAleph body
<[700]> evalsupport module end
<[1]> evaltime_meta module start
<[2]> ClassThree body
<[200]> deco_alpha
<[4]> ClassFour body
<[6]> ClassFive body
<[500]> MetaAleph.__init__
<[9]> ClassSix body
<[500]> MetaAleph.__init__
<[11]> ClassThree tests ..............................
<[300]> deco_alpha:inner_1 ❶
<[12]> ClassFour tests ..............................
<[5]> ClassFour.method_y ❷
<[13]> ClassFive tests ..............................
<[7]> ClassFive.__init__
<[600]> MetaAleph.__init__:inner_2 ❸
<[14]> ClassSix tests ..............................
<[7]> ClassFive.__init__
<[600]> MetaAleph.__init__:inner_2 ❹
<[15]> evaltime_meta module end
```

❶ 装饰器依附到 ClassThree 类上之后，method_y 方法被替换成 inner_1 方法……

❷ 虽然 ClassFour 是 ClassThree 的子类，但是没有依附装饰器的 ClassFour 类却不受影响。

❸ MetaAleph 类的 __init__ 方法把 ClassFive.method_z 方法替换成 inner_2 函数。

❹ ClassFive 的子类 ClassSix 也是一样，method_z 方法被替换成 inner_2 函数。

注意，ClassSix 类没有直接引用 MetaAleph 类，但是却受到了影响，因为它是 ClassFive 的子类，进而也是 MetaAleph 类的实例，所以由 MetaAleph.__init__ 方法初始化。

> 如果想进一步定制类，可以在元类中实现 __new__ 方法。不过，通常情况下实现 __init__ 方法就够了。

现在，我们可以实践这些理论了。我们将创建一个元类，让描述符以最佳的方式自动创建储存属性的名称。

21.5　定制描述符的元类

回到 `LineItem` 系列示例。如果用户完全不用知道描述符或元类，直接继承库提供的类就能满足需求，那该多好。如示例 21-14 所示。

示例 **21-14**　bulkfood_v7.py：有元类的支持，继承 `model.Entity` 类即可

```
import model_v7 as model

class LineItem(model.Entity):  ❶
    description = model.NonBlank()
    weight = model.Quantity()
    price = model.Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ `LineItem` 是 `model.Entity` 的子类。

示例 21-14 理解起来相当容易，毕竟根本没有奇怪的句法。可是，model_v7.py 模块必须定义一个元类，而且 `model.Entity` 类是那个元类的实例。model_v7.py 模块中实现的 `Entity` 类如示例 21-15 所示。

示例 **21-15**　model_v7.py：`EntityMeta` 元类以及它的一个实例 `Entity`

```
class EntityMeta(type):
    """元类，用于创建带有验证字段的业务实体"""

    def __init__(cls, name, bases, attr_dict):
        super().__init__(name, bases, attr_dict)  ❶
        for key, attr in attr_dict.items():  ❷
            if isinstance(attr, Validated):
```

```
                type_name = type(attr).__name__
                attr.storage_name = '_{}#{}'.format(type_name, key)

class Entity(metaclass=EntityMeta):  ❸
    """带有验证字段的业务实体"""
```

❶ 在超类（在这里是 `type`）上调用 `__init__` 方法。

❷ 与示例 21-4 中 `@entity` 装饰器的逻辑一样。

❸ 这个类的存在只是为了用起来便利：这个模块的用户直接继承 `Entity` 类即可，无需关心 `EntityMeta` 元类，甚至不用知道它的存在。

示例 21-14 中的代码能通过示例 21-3 中的测试。辅助模块 model_v7.py 比 model_v6.py 难理解，但是用户级别的代码更简单：只需继承 `model_v7.Entity` 类，`Validated` 字段就能自动获得储存属性的名称。

图 21-4 使用简单的图示说明了我们刚刚实现的逻辑。虽然有很多复杂的逻辑，但都隐藏在 `model_v7` 模块中。从用户的角度来看，示例 21-14 中的 `LineItem` 只是 `Entity` 的子类。这就是抽象的作用。



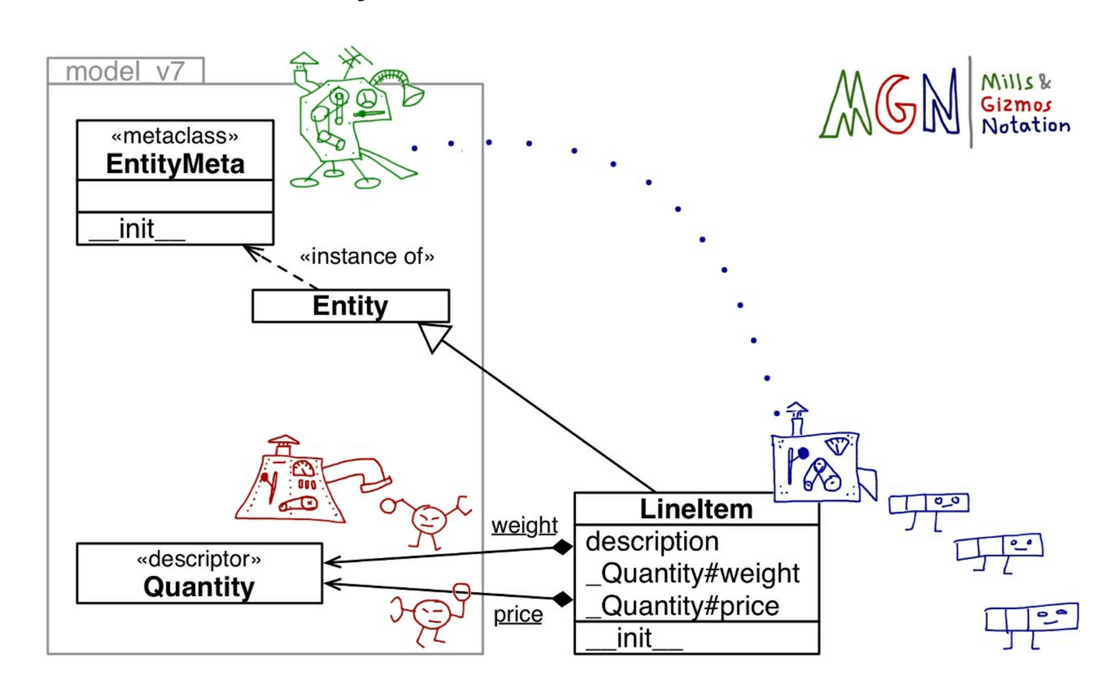


图 **21-4**：使用机器和小怪兽图示法（**MGN**）注解的 **UML** 类图。**`EntityMeta`** 元机器用于生产 **`LineItem`** 机器。描述符（如 **`weight`** 和 **`price`**）由 **`EntityMeta.__init__`** 方法配置。注意 **`model_v7`** 模块的边界

除了把类链接到元类上的句法之外 [5]，目前编写元类使用的句法在 Python 2.2（这个版本对 Python 类型做了重大改造）之后都能使用。下一节介绍一个只能在 Python 3 中使用的功能。

[5] 11.7.1 节说过，Python 2.7 使用的是 `__metaclass__` 类属性，类的声明体不支持 `metaclass=` 关键字参数。

21.6　元类的特殊方法 `__prepare__`

在某些应用中，可能需要知道类的属性定义的顺序。例如，对读写 CSV 文件的库来说，用户定义的类可能想把类中按顺序声明的字段与 CSV 文件中各列的顺序对应起来。

如前所述，`type` 构造方法及元类的 `__new__` 和 `__init__` 方法都会收到要计算的类的定义体，形式是名称到属性的映像。然而在默认情况下，那个映射是字典；也就是说，元类或类装饰器获得映射时，属性在类定义体中的顺序已经丢失了。

这个问题的解决办法是，使用 Python 3 引入的特殊方法 `__prepare__`。这个特殊方法只在元类中有用，而且必须声明为类方法（即，要使用 `@classmethod` 装饰器定义）。解释器调用元类的 `__new__` 方法之前会先调用 `__prepare__` 方法，使用类定义体中的属性创建映射。`__prepare__` 方法的第一个参数是元类，随后两个参数分别是要构建的类的名称和基类组成的元组，返回值必须是映射。元类构建新类时，`__prepare__` 方法返回的映射会传给 `__new__` 方法的最后一个参数，然后再传给 `__init__` 方法。

理论听起来很复杂，但是我见过的 `__prepare__` 方法都十分简单。请看示例 21-16。

示例 **21-16**　model_v8.py：这一版 `EntityMeta` 元类用到了 `__prepare__` 方法，而且为 `Entity` 类定义了 `field_names` 类方法

```
class EntityMeta(type):
    """元类，用于创建带有验证字段的业务实体"""

    @classmethod
    def __prepare__(cls, name, bases):
        return collections.OrderedDict()  ❶

    def __init__(cls, name, bases, attr_dict):
        super().__init__(name, bases, attr_dict)
        cls._field_names = []  ❷
```

```
        for key, attr in attr_dict.items():  ❸
            if isinstance(attr, Validated):
                type_name = type(attr).__name__
                attr.storage_name = '_{}#{}'.format(type_name, key)
                cls._field_names.append(key)  ❹


class Entity(metaclass=EntityMeta):
    """带有验证字段的业务实体"""

    @classmethod
    def field_names(cls):  ❺
        for name in cls._field_names:
            yield name
```

❶ 返回一个空的 `OrderedDict` 实例，类属性将存储在里面。

❷ 在要构建的类中创建一个 `_field_names` 属性。

❸ 这一行与前一版相比没有变化，不过这里的 `attr_dict` 是那个 `OrderedDict` 对象，由解释器在调用 `__init__` 方法之前调用 `__prepare__` 方法时获得。因此，这个 `for` 循环会按照添加属性的顺序迭代属性。

❹ 把找到的各个 `Validated` 字段添加到 `_field_names` 属性中。

❺ `field_names` 类方法的作用简单：按照添加字段的顺序产出字段的名称。

像示例 21-16 那样添加一些简单的代码之后，我们可以使用 `field_names` 类方法迭代任何 `Entity` 子类的 `Validated` 字段。示例 21-17 演示了这个新功能。

示例 **21-17**　bulkfood_v8.py：展示 `field_names` 用法的 doctest——无需修改 `LineItem` 类，`field_names` 方法继承自 `model.Entity` 类

```
>>> for name in LineItem.field_names():
...     print(name)
...
```

```
description
weight
price
```

对元类的介绍到此结束。在现实世界中，框架和库会使用元类协助程序员执行很多任务，例如：

- 验证属性
- 一次把装饰器依附到多个方法上
- 序列化对象或转换数据
- 对象关系映射
- 基于对象的持久存储
- 动态转换使用其他语言编写的类结构

下一节将概述 Python 数据模型为所有类定义的方法。

21.7　类作为对象

Python 数据模型为每个类定义了很多属性，参见标准库参考中“Built-in Types”一章的“4.13. Special Attributes”一节（https://docs.python.org/3/library/stdtypes.html#special-attributes）。其中三个属性在本书中已经见过多次：`__mro__`、`__class__` 和 `__name__`。此外，还有以下属性。

`cls.__bases__`

　　由类的基类组成的元组。

`cls.__qualname__`

　　Python 3.3 新引入的属性，其值是类或函数的限定名称，即从模块的全局作用域到类的点分路径。例如，在示例 21-6 中，内部类 `ClassTwo` 的 `__qualname__` 属性，其值是字符串 `'ClassOne.ClassTwo'`，而 `__name__` 属性的值是 `'ClassTwo'`。这个属性的规范是“PEP 3155—Qualified name for classes and functions”（https://www.python.org/dev/peps/pep-3155/）。

`cls.__subclasses__()`

　　这个方法返回一个列表，包含类的直接子类。这个方法的实现使用弱引用，防止在超类和子类（子类在 `__bases__` 属性中储存指向超类的强引用）之间出现循环引用。这个方法返回的列表中是内存里现存的子类。

`cls.mro()`

　　构建类时，如果需要获取储存在类属性 `__mro__` 中的超类元组，解释器会调用这个方法。元类可以覆盖这个方法，定制要构建的类解析方法的顺序。



`dir(...)` 函数不会列出本节提到的任何一个属性。

我们对类元编程的学习到此结束。这是个很大的话题，我只讲了皮毛。因此，本书各章都有“延伸阅读”一节。

21.8　本章小结

类元编程是指动态创建或定制类。在 Python 中，类是一等对象，因此本章首先说明如何通过调用内置的 `type` 元类，使用函数创建类。

接下来的一节继续讨论第 20 章使用描述符实现的 `LineItem` 类，解决一个遗留问题：如何让生成的储存属性名中包含托管属性的名称（例如，把 `_Quantity#1` 变成 `_Quantity#price`）。解决办法是使用类装饰器。说到底，类装饰器是函数，其参数是被装饰的类，用于审查和修改刚创建的类，甚至替换成其他类。

然后，本章讨论了模块中不同部分的代码何时运行。我们发现，所谓的“导入时”和“运行时”之间有重叠，不过很明显，`import` 语句会触发运行大量代码。知道代码何时运行至关重要，可是有些规则难以捉摸，因此我们通过两个计算时间练习对此做了说明。

接下来，本章介绍了元类。我们得知，所有类都直接或间接地是 `type` 的实例，因此在 Python 中，`type` 是“根元类”。然后，我们对之前的计算时间练习做了修改，以此说明元类可以定制类的层次结构。类装饰器则不同，它只能影响一个类，而且对后代可能没有影响。

随后，我们实际使用元类，解决 `LineItem` 类中储存属性的命名问题。最终写出的代码比类装饰器难懂一些，不过可以封装在一个模块里，这样用户只需继承看似普通的一个类（`model.Entity`），而不用知道它是元类（`model.EntityMeta`）的实例。这种处理方式让人想起了 Django 和 SQLAlchemy 的 ORM API：使用元类实现，用户却根本无需知道。

我们实现的第二个元类为 `model.EntityMeta` 类添加了一个小功能：定义 `__prepare__` 方法，返回一个 `OrderedDict` 对象，用于储存名称到属性的映射。这样做能保留要构建的类在定义体中绑定属性的顺序，提供给元类的 `__new__` 和 `__init__` 等方法使用。在这个示例中，我们定义了类属性 `_field_names`，因此用户可以使用 `Entity.field_names()` 方法以 `Validated` 描述符出现在源码中的顺序获取描述符。

最后一节，我们概述了 Python 为所有类提供的属性和方法。

元类是充满挑战、让人兴奋的功能，有时会被故作聪明的程序员滥用。最后，我们回顾一下 Alex Martelli 在他写的“水禽和抽象基类”一文的最后给我们的建议：

> 此外，不要在生产代码中定义抽象基类（或元类）……如果你很想这样做，我打赌可能是因为你想“找茬”，刚拿到新工具的人都有大干一场的冲动。如果你能避开这些深奥的概念，你（以及未来的代码维护者）的生活将更愉快，因为代码简洁明了。
>
> ——Alex Martelli

说出上述至理名言的人不仅是 Python 元编程大师，还是造诣颇深的软件工程师，负责世界上几个最重要的 Python 应用。

21.9　延伸阅读

为了深入学习本章所述的知识，一定要阅读 Python 语言参考手册中“Data Model”一章里的“3.3.3. Customizing class creation”一节（https://docs.python.org/3/reference/datamodel.html#metaclasses）、“Built-in Functions”一章中 `type` 类的文档（https://docs.python.org/3/library/functions.html#type），以及标准库参考中“Built-in Types”一章里的“4.13. Special Attributes”一节（https://docs.python.org/3/library/stdtypes.html#special-attributes）。此外，在标准库参考中，`types` 模块的文档（https://docs.python.org/3/library/types.html）说明了 Python 3.3 引入的两个新函数，这两个函数用于辅助类元编程：`types.new_class(...)` 和 `types. prepare_class(...)`。

类装饰器的规范是“PEP 3129—Class Decorators”（https://www.python.org/dev/peps/pep-3129/），作者是 Collin Winter，参考实现由 Jack Diederich 提供。Jack Diederich 在 PyCon 2009 大会上做了一场题为“Class Decorators: Radically Simple”的演讲（视频：https://www.youtube.com/watch?v=cAGliEJV9_o），对这个功能做了简单介绍。

Alex Martelli 写的《Python 技术手册（第 2 版）》对元类的说明很出色，还实现了 `metaMetaBunch` 元类，其作用与示例 21-2 中简单的 `record_factory` 函数一样，不过完善得多。Martelli 没有探讨类装饰器，因为这个功能在那本书出版后才引入。Beazley 和 Jones 在他们合著的《Python Cookbook（第 3 版）中文版》中提供了几个示例，很好地演示了类装饰器和元类。Michael Foord 写了一篇引人入胜的文章，题为“Meta-classes Made Easy: Eliminating self with Metaclasses”（http://www.voidspace.org.uk/python/articles/metaclasses.shtml 副标题（“借助元类去掉 self”）说明了一切。

元类的主要参考资料有引入特殊方法 `__prepare__` 的“PEP 3115—Metaclasses in Python 3000”（https://www.python.org/dev/peps/pep-3115/），以及 Guido van Rossum 发布的文章“Unifying types and classes in Python

2.2”（https://www.python.org/download/releases/2.2.3/descrintro/）。这篇文章也适用于 Python 3，谈到了后来称为“新式类”的语义，包括描述符和元类，一定要阅读。Guido 在文中提到了 Ira R. Forman 与 Scott H. Danforth 合著的 *Putting Metaclasses to Work: a New Dimension in Object-Oriented Programming*（Addison- Wesley 出版社，1998 年），他在亚马逊上给这本书打了五颗星，还写了如下评论：

> 这本书促成 **Python 2.2** 实现了元类
>
> 可惜，这本书已经绝版了。Python 通过 `super()` 函数实现了协作式多重继承，谈到这方面的难题时，我总会提到这本书；据我所知，这本书是这方面最好的教程。[6]

> [6]摘自亚马逊网站中 *Putting Metaclasses to Work* 的商品目录页面（http://amzn.to/1HGwKDO）。目前还有二手书出售。我买了一本，发现很难读懂，不过以后我可能会再读。

“PEP 487—Simpler customization of class creation”（https://www.python.org/dev/peps/pep-0487/）提议为 Python 3.5（写到这里时，处于内测阶段）添加一个新的特殊方法 `__init_subclass__`，[7] 让普通的类（即，不是元类）定制子类的初始化。与类装饰器一样，`__init_subclass__` 方法能让类元编程变得更简单，但会导致元类这个强大的功能更难正确使用。

> [7]现在，Python 3.5 已经正式发布，PEP 487 没有在 Python 3.5 中实现，而是推迟到 Python 3.6 中。——编者注

如果你喜欢元编程，可能希望 Python 提供基本的元编程功能——Elixir 和 Lisp 语言族提供的句法宏。天遂人愿，我们有 MacroPy（https://github.com/lihaoyi/macropy）可用。

杂谈

> 这是本书最后一篇“杂谈”了，首先我要从 Brian Harvey 与 Matthew Wright 合写的著作中引述一大段文字。Harvey 和 Wright 是加州大学（伯克利分校和圣巴巴拉分校）的计算机科学教授，他们在合著的 *Simply Scheme* 一书中写道：

计算机科学的教学方式分成两个流派，可以描述如下。

> (1) 保守派 计算机程序已经变得极其大而复杂，超过了人类思维所能承载的限度。因此，计算机科学教育的任务是训练平庸的程序员，这样 500 个人合作便能开发出恰好满足需求的程序。
>
> (2)激进派 计算机程序已经变得极其大而复杂，超过了人类思维所能承载的限度。因此，计算机科学教育的任务是教人如何拓展思维，打破常规，学习以更广博、更强大和更灵活的方式思考，让思维超越程序。编程思想的各个方面在程序中必会得到充分体现。8[Brian Harvey and Matthew Wright, *Simply Scheme* (MIT Press, 1999), p. xvii. 伯克利分校的网站中有此书全文（https://www.eecs.berkeley.edu/~bh/ss-toc2.html）。]}
>
> ——Brian Harvey 和 Matthew Wright
>
> *Simply Scheme* 前言

这是 Harvey 和 Wright 对计算机科学教育的夸张描述，不过也适用于编程语言的设计。现在，你应该能猜到，我赞成“激进派”，我认为 Python 也是以这种态度设计的。

为了稳扎稳打，Java 从一开始使用的就是存取方法，而且众多 Java IDE 都提供了生成读值方法和设值方法的快捷键；与此相比，特性算是一大进步。特性的主要优点是，一开始编写程序时可以先把属性设为公开的（遵照 KISS 原则），因为公开的属性无需大幅改动，随时都能变成特性。不过，描述符更进一步，提供了去除存取方法中逻辑重复的机制。这种机制特别有效，因此基本的 Python 结构在背后也用到了描述符。

另一个强大的想法是，把函数当作一等对象，这为高阶函数铺平了道路。描述符和高阶函数合在一起实现，使得函数和方法的统一成为可能。函数的 `__get__` 方法能即时生成方法对象，把实例绑定到 `self` 参数上。这种做法相当优雅。[9]

最后，Python 中的类也是一等对象。作为一门对初学者友好的语言，Python 能提供类装饰器，允许用户定义功能完整的元类，这些强大的抽象真是太棒了。最棒的是，这些高级功能没有拖累日常编

程（其实无形中提供了帮助）。Django 和 SQLAlchemy 等框架用起来这么方便，发展得这么成功，很大程度上归功于元类，而这些工具的用户甚至不知道元类的存在。不过，他们可以学习，去创建下一个伟大的库。

我还未见过有哪门语言像 Python 这样竭尽所能，让初学者易于入门，让专业人士用着顺手，让程序高手欢欣鼓舞。感谢 Guido van Rossum，以及为此努力的每个人。

> [9]David Gelernter 写的 *Machine Beauty*（Basic Books 出版社）是一本非常有趣的小书，对工程作品（从桥梁到软件）的优雅和美学做了阐述。

结语

> Python 是给法定成年人使用的语言。
>
> ——Alan Runyan
> Plone 的联合创始人

Alan 的精辟定义道出了 Python 最好的特质之一：它不妨碍你，让你做你该做的事。这也意味着，它不会给你提供工具，让你限制其他人能对你的代码和代码所构建的对象做什么。

当然，Python 不完美。对我来说，最没法接受的是，Python 在标准库中混用驼峰式和蛇底式，或者直接把单词连在一起。但是，语言的定义和标准库只是生态系统的一部分。用户和贡献者组成的社区才是 Python 生态系统最重要的部分。

有一个例子可以说明社区的好处。一天早上，我在撰写 `asyncio` 包相关的内容时，感到很沮丧，因为那个包的 API 有很多函数，其中有些是协程，可是协程必须使用 `yield from` 调用，而常规的函数不能这么做。这在 `asyncio` 包的文档中有说明，可是有时阅读几段文字之后才能确定某个函数是不是协程。因此，我给 python-tulip 邮件列表发了一个消息，题为“Proposal: make coroutines stand out in the asyncio docs”（https://groups.google.com/forum/#!topic/python-tulip/Y4bhLNbKs74）。`asyncio` 包的核心开发者 Victor Stinner、`aiohttp` 包的主要作者 Andrew Svetlov、Tornado 的首席开发者 Ben Darnell，以及 Twisted 的发明者 Glyph Lefkowitz 加入了讨论。Darnell 提出了一个方案，Alexander Shorin 解说如何在 Sphinx 中实现，Stinner 添加了所需的配置和标记。我提出这个问题不到 12 小时，`asyncio` 包的整个线上文档都更新了，添加了今天你所看到的“coroutine”标签（https://docs.python.org/3/library/asyncio-eventloop.html#executor）。

在排外的社区中绝不会有这种事。任何人都能加入 python-tulip 邮件列表，我编写那个提议之前只发布过几次消息而已。这个故事表明，Python 社区特别开放，广纳新想法和新成员。Guido van Rossum 也在

python-tulip 邮件列表中，即使是简单的问题也经常回答。

还有一个例子能说明 Python 的开放：Python 软件基金会（Python Software Foundation，PSF）一直在努力提升 Python 社区的多样性，而且已经达成一些令人欣喜的成果。2013— 2014 年，PSF 董事会首次选出了女性董事——Jessica McKellar 和 Lynn Root。2015 年在蒙特利尔举办的 PyCon North America 大会（Diana Clarke 主持），约 1/3 的演讲者是女性。我还没见过其他 IT 大会如此追求性别平等。

如果你是 Python 程序员，但尚未加入社区，我建议你快点加入。寻找你所在地区的 Python 用户组（Python Users Group，PUG）。如果没有，那就创建一个。任何地方都有人使用 Python，你并不孤独。如果可能的话，参加别处举办的会议。来参加 PythonBrasil 大会吧，多年以来这个大会都有来自世界各地的演讲者。与其他 Python 程序员见面比任何线上互动都好，除了可以获得别人分享的知识外，还有很多好处，例如工作机会和真正的友谊。

我知道，如果没有多年来我在 Python 社区中结交的朋友的帮助，我不可能写出这本书。

我的父亲说过，“Só erra quem trabalha”，这是葡萄牙语，意思是“只有真正做事的人才会犯错”。这个建议很棒，能让你不再害怕失败，迈步向前。撰写这本书的过程中，我肯定犯了错误。审校、编辑和预先发布版的读者帮我找出了很多错误。早期发布版刚发布几小时，就有一个读者在本书的勘误页面（http://www.oreilly.com/catalog/errata.csp?isbn=0636920032519）报告拼写错误。其他读者报告了更多错误，我的朋友还直接联系我，提供建议和更正。我写完本书后，O'Reilly 的文字编辑会在出版过程中找出其他错误。如果还有任何错误和词不达意的表述，责任都在我，在此向各位读者致歉。

终于写完这本书了，我特别高兴，无论有没有错误，我都十分感激一路上给我帮助的每个人。

希望很快就能在会议上见到你。如果见到我，请过来打声招呼。

延伸阅读

在本书的最后，我要介绍一些“Python 风格”的参考资料——这正是本书尝试解决的主要问题。

Brandon Rhodes 是位出色的 Python 教师，他的演讲“A Python Æsthetic: Beauty and Why I Python”（https://www.youtube.com/watch?v=x-kB2o8sd5c）很精彩，从标题中使用的 Unicode 字符 U+00C6（拉丁语大写字母 AE）开始谈起。另一位出色的教师 Raymond Hettinger，在 2013 年的 PyCon US 大会上谈了 Python 之美：“Transforming Code into Beautiful, Idiomatic Python”（https://www.youtube.com/watch?v=OSGv2VnC0go）。

Ian Lee 在 Python-ideas 邮件列表中发起的“Evolution of Style Guides”话题（https://mail.python.org/pipermail/python-ideas/2015-March/032557.html）值得一读。Lee 是 `pep8` 包（https://pypi.python.org/pypi/pep8/）的维护者，这个包的作用是检查 Python 代码是否符合 PEP 8。检查书中的代码时，我用的是 `flake8`（https://pypi.python.org/pypi/flake8），这个包融合了 `pep8`、`pyflakes`（https://pypi.python.org/pypi/pyflakes）和 Ned Batchelder 开发的 McCabe 复杂度插件（https://pypi.python.org/pypi/mccabe）。

除了 PEP 8，Google 的 Python 风格指南（https://google-styleguide.googlecode.com/svn/trunk/pyguide.html）和 Pocoo 风格指南（http://www.pocoo.org/internal/styleguide/）也有很大的影响。Pocoo 团队为我们开发了 Flask、Sphinx、Jinja 2 和其他优秀的 Python 库。

The Hitchhiker's Guide to Python!（http://docs.python-guide.org/en/latest/）由多人维护，说明如何编写符号 Python 风格的代码。为这个项目贡献最多内容的是 Kenneth Reitz，他因开发特别符合 Python 风格的 `requests` 包而被社区视为英雄。David Goodger 在 2008 年举办的 PyCon US 大会上办了一场教学活动，题为“Code Like a Pythonista: Idiomatic Python”（http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.ht
如果打印出来，这个教程的教案有 30 页。当然，教案的 reStructuredText 源码能下载到，可以使用 `docutils` 将其渲染成 HTML

和 S5 幻灯片（http://meyerweb.com/eric/tools/s5/）。毕竟，reStructuredText 和 `docutils` 都是 Goodger 的作品。这两个工具是 Sphinx 的基础。Sphinx 是优秀的 Python 文档系统，顺便提一下，MongoDB（https://docs.mongodb.org/manual/about/#about-the-documentation-process）和很多其他项目的官方文档系统都是 Sphinx。

Martijn Faassen 直接回答了“什么是 Python 风格”这个问题（http://blog.startifact.com/posts/older/what-is-pythonic.html），python-list 邮件列表中也有一个相同标题的话题（https://mail.python.org/pipermail/tutor/2003-October/025930.html）。Martijn 的文章是 2005 年写的，那个话题是 2003 年讨论的，不过 Python 风格的思想没怎么变化，Python 语言本身也是如此。“Pythonic way to sum n-th list element?”话题（https://mail.python.org/pipermail/python-list/2003-April/192027.html）对 Python 风格做了深入讨论，我在第 10 章的“杂谈”中有大量引用。

“PEP 3099 — Things that will Not Change in Python 3000”（https://www.python.org/dev/peps/pep-3099/）解释了经过 Python 3 大幅度的调整之后，为何许多东西仍是现在的样子。长久以来，Python 3 有个昵称——Python 3000，不过诞生时间早了几个世纪，这让一些人失望。PEP 3099 的作者是 Georg Brandl，他收集了仁慈的独裁者（即 Guido van Rossum）的很多观点。Python Essays 页面（https://www.python.org/doc/essays/）列出了很多 Guido 自己写的文章。

附录 A　辅助脚本

有些脚本太长，在正文里放不下，这里将其完整列出。此外，有些脚本用于生成书中的表格和数据，这里一并列出。

这里列出的脚本，以及书中几乎每个代码片段，见于本书的代码仓库（https://github.com/fluentpython/example-code）。

A.1　第3章：in运算符的性能测试

表 3-6 中的计时数据是我使用示例 A-1 中的代码生成的，这段代码用到了 timeit 模块。这个脚本主要用于设置 haystack 和 needles 样本，并格式化输出。

编写示例 A-1 时，我发现的确能客观比较 dict 的性能。如果在“详细模式”（指定命令行选项 -v）中运行这个脚本，用时几乎是表 3-5 中的两倍。但是注意，对这个脚本来说，在“详细模式”中，只是多了用于设置测试内容的四个 print 调用，以及在各个测试结束后显示找到多少个 needles 的那个 print 调用。在 haystack 中搜索 needles 的那个循环没有输出，不过这五个 print 调用耗费的时间与搜索 1000 个 needles 差不多。

示例 **A-1**　container_perftest.py：运行时以内置集合类型的名称为命令行参数（例如 container_perftest.py dict）

```
"""
对容器的``in``运算符做性能测试
"""
import sys
import timeit

SETUP = '''
import array

selected = array.array('d')
with open('selected.arr', 'rb') as fp:
    selected.fromfile(fp, {size})
if {container_type} is dict:
    haystack = dict.fromkeys(selected, 1)
else:
    haystack = {container_type}(selected)
if {verbose}:
    print(type(haystack), end='  ')
    print('haystack: %10d' % len(haystack), end='  ')
needles = array.array('d')
with open('not_selected.arr', 'rb') as fp:
    needles.fromfile(fp, 500)
needles.extend(selected[::{size}//500])
```

```
if {verbose}:
    print(' needles: %10d' % len(needles), end='  ')
'''

TEST = '''
found = 0
for n in needles:
    if n in haystack:
        found += 1
if {verbose}:
    print('  found: %10d' % found)
'''

def test(container_type, verbose):
    MAX_EXPONENT = 7
    for n in range(3, MAX_EXPONENT + 1):
        size = 10**n
        setup = SETUP.format(container_type=container_type,
                             size=size, verbose=verbose)
        test = TEST.format(verbose=verbose)
        tt = timeit.repeat(stmt=test, setup=setup, repeat=5, number=1)
        print('|{:{}d}|{:f}'.format(size, MAX_EXPONENT + 1, min(tt)))

if __name__=='__main__':
    if '-v' in sys.argv:
        sys.argv.remove('-v')
        verbose = True
    else:
        verbose = False
    if len(sys.argv) != 2:
        print('Usage: %s <container_type>' % sys.argv[0])
    else:
        test(sys.argv[1], verbose)
```

container_perftest_datagen.py 脚本（见示例 A-2）为示例 A-1 中的脚本生成固件数据。

示例 **A-2**　container_perftest_datagen.py：生成由不同的浮点数组成的数组，然后写入文件，供示例 A-1 使用

```
"""
生成容器性能测试所需的数据
"""
```

```python
import random
import array

MAX_EXPONENT = 7
HAYSTACK_LEN = 10 ** MAX_EXPONENT
NEEDLES_LEN = 10 ** (MAX_EXPONENT - 1)
SAMPLE_LEN = HAYSTACK_LEN + NEEDLES_LEN // 2

needles = array.array('d')

sample = {1/random.random() for i in range(SAMPLE_LEN)}
print('initial sample: %d elements' % len(sample))

# 完整的样本，防止丢弃了重复的随机数
while len(sample) < SAMPLE_LEN:
    sample.add(1/random.random())

print('complete sample: %d elements' % len(sample))

sample = array.array('d', sample)
random.shuffle(sample)

not_selected = sample[:NEEDLES_LEN // 2]
print('not selected: %d samples' % len(not_selected))
print('  writing not_selected.arr')
with open('not_selected.arr', 'wb') as fp:
    not_selected.tofile(fp)

selected = sample[NEEDLES_LEN // 2:]
print('selected: %d samples' % len(selected))
print('  writing selected.arr')
with open('selected.arr', 'wb') as fp:
    selected.tofile(fp)
```

A.2　第3章：比较散列后的位模式

示例 A-3 是个简单的脚本，告诉你相似浮点数（例如 1.0001、1.0002，等等）的位模式有什么差异。这个脚本的输出在示例 3-16 中。

示例 **A-3**　hashdiff.py：显示散列值的位模式有何差异

```
import sys

MAX_BITS = len(format(sys.maxsize, 'b'))
print('%s-bit Python build' % (MAX_BITS + 1))

def hash_diff(o1, o2):
    h1 = '{:>0{}b}'.format(hash(o1), MAX_BITS)
    h2 = '{:>0{}b}'.format(hash(o2), MAX_BITS)
    diff = ''.join('!' if b1 != b2 else ' ' for b1, b2 in zip(h1, h2))
    count = '!= {}'.format(diff.count('!'))
    width = max(len(repr(o1)), len(repr(o2)), 8)
    sep = '-' * (width * 2 + MAX_BITS)
    return '{!r:{width}} {}\n{:{width}} {} {}\n{!r:{width}} {}\n{}'.format(
                o1, h1, ' ' * width, diff, count, o2, h2, sep, width=width)

if __name__ == '__main__':
    print(hash_diff(1, 1.0))
    print(hash_diff(1.0, 1.0001))
    print(hash_diff(1.0001, 1.0002))
    print(hash_diff(1.0002, 1.0003))
```

A.3 第9章：有或没有 `__slots__` 时，RAM的用量

memtest.py 脚本用于支持 9.8 节的一个演示——示例 9-12。

memtest.py 脚本从命令行中接收一个模块的名称，加载那个模块。假设模块中定义有一个名为 `Vector` 的类，memtest.py 脚本会创建一个由一千万个实例组成的列表，然后报告创建列表前后内存的用量。

示例 **A-4** memtest.py：创建大量 `Vector` 实例，报告内存用量

```
import importlib
import sys
import resource

NUM_VECTORS = 10**7

if len(sys.argv) == 2:
    module_name = sys.argv[1].replace('.py', '')
    module = importlib.import_module(module_name)
else:
    print('Usage: {} <vector-module-to-test>'.format())
    sys.exit(1)

fmt = 'Selected Vector2d type: {.__name__}.{.__name__}'
print(fmt.format(module, module.Vector2d))

mem_init = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss
print('Creating {:,} Vector2d instances'.format(NUM_VECTORS))

vectors = [module.Vector2d(3.0, 4.0) for i in range(NUM_VECTORS)]

mem_final = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss
print('Initial RAM usage: {:14,}'.format(mem_init))
print('  Final RAM usage: {:14,}'.format(mem_final))
```

A.4　第14章：转换数据库的isis2json.py脚本

示例 A-5 是 14.13 节讨论的 isis2json.py 脚本。这个脚本使用生成器函数，以惰性的方式把 CDS/ISIS 数据库转换成 JSON 格式，以便载入到 CouchDB 或 MongoDB。

注意，这是个 Python 2 脚本，针对 CPython 或 Jython，支持 Python 2.5~2.7，不能使用 Python 3 运行。在 CPython 中，只能读取 .iso 文件；在 Jython 中，使用 GitHub 中 fluentpython/isis2json 仓库（https://github.com/fluentpython/isis2json）里的 `Bruma` 库，还可以读取 .mst 文件。详情参见该仓库里的用法文档。

示例 **A-5**　isis2json.py：依赖和文档在 GitHub 中的 fluentpython/isis2json 仓库里

```
# 这个脚本支持Python和Jython（版本>=2.5且&lt;3）

import sys
import argparse
from uuid import uuid4
import os

try:
    import json
except ImportError:
    if os.name == 'java':  # 在Jython中运行
        from com.xhaus.jyson import JysonCodec as json
    else:
        import simplejson as json

SKIP_INACTIVE = True
DEFAULT_QTY = 2**31
ISIS_MFN_KEY = 'mfn'
ISIS_ACTIVE_KEY = 'active'
SUBFIELD_DELIMITER = '^'
INPUT_ENCODING = 'cp1252'


def iter_iso_records(iso_file_name, isis_json_type):  ❶
```

```
    from iso2709 import IsoFile
    from subfield import expand

    iso = IsoFile(iso_file_name)
    for record in iso:
        fields = {}
            for field in record.directory:
            field_key = str(int(field.tag)) # 删除前导零
            field_occurrences = fields.setdefault(field_key, [])
            content = field.value.decode(INPUT_ENCODING, 'replace')
            if isis_json_type == 1:
                field_occurrences.append(content)
            elif isis_json_type == 2:
                field_occurrences.append(expand(content))
            elif isis_json_type == 3:
                field_occurrences.append(dict(expand(content)))
            else:
                raise NotImplementedError('ISIS-JSON type %s conversion '
                    'not yet implemented for .iso input' % isis_json_type)

        yield fields
    iso.close()


def iter_mst_records(master_file_name, isis_json_type):  ❷
    try:
        from bruma.master import MasterFactory, Record
    except ImportError:
        print('IMPORT ERROR: Jython 2.5 and Bruma.jar '
              'are required to read .mst files')
        raise SystemExit
    mst = MasterFactory.getInstance(master_file_name).open()
    for record in mst:
        fields = {}
        if SKIP_INACTIVE:
            if record.getStatus() != Record.Status.ACTIVE:
                continue
        else:  # 仅当没有活动的记录时才保存状态
            fields[ISIS_ACTIVE_KEY] = (record.getStatus() ==
                                       Record.Status.ACTIVE)
        fields[ISIS_MFN_KEY] = record.getMfn()
        for field in record.getFields():
            field_key = str(field.getId())
            field_occurrences = fields.setdefault(field_key, [])
            if isis_json_type == 3:
                content = {}
                for subfield in field.getSubfields():
```

```
                    subfield_key = subfield.getId()
                    if subfield_key == '*':
                        content['_'] = subfield.getContent()
                    else:
                        subfield_occurrences = content.setdefault(subfield_ke
                        subfield_occurrences.append(subfield.getContent())
                field_occurrences.append(content)
            elif isis_json_type == 1:
                content = []
                for subfield in field.getSubfields():
                    subfield_key = subfield.getId()
                    if subfield_key == '*':
                        content.insert(0, subfield.getContent())
                    else:
                        content.append(SUBFIELD_DELIMITER + subfield_key +
                                       subfield.getContent())
                field_occurrences.append(''.join(content))
            else:
                raise NotImplementedError('ISIS-JSON type %s conversion '
                    'not yet implemented for .mst input' % isis_json_type)
        yield fields
    mst.close()


def write_json(input_gen, file_name, output, qty, skip, id_tag,  ❸
               gen_uuid, mongo, mfn, isis_json_type, prefix,
               constant):
    start = skip
    end = start + qty
    if id_tag:
        id_tag = str(id_tag)
        ids = set()
    else:
        id_tag = ''
    for i, record in enumerate(input_gen):
        if i >= end:
            break
        if not mongo:
            if i == 0:
                output.write('[')
            elif i > start:
                output.write(',')
        if start <= i < end:
            if id_tag:
                occurrences = record.get(id_tag, None)
                if occurrences is None:
                    msg = 'id tag #%s not found in record %s'
```

```
                if ISIS_MFN_KEY in record:
                    msg = msg + (' (mfn=%s)' % record[ISIS_MFN_KEY])
                raise KeyError(msg % (id_tag, i))
            if len(occurrences) > 1:
                msg = 'multiple id tags #%s found in record %s'
                if ISIS_MFN_KEY in record:
                   msg = msg + (' (mfn=%s)' % record[ISIS_MFN_KEY])
                raise TypeError(msg % (id_tag, i))
            else:  # 好吧，有且仅有一个id字段
                if isis_json_type == 1:
                    id = occurrences[0]
                elif isis_json_type == 2:
                    id = occurrences[0][0][1]
                elif isis_json_type == 3:
                    id = occurrences[0]['_']
                if id in ids:
                    msg = 'duplicate id %s in tag #%s, record   %s'
                    if ISIS_MFN_KEY in record:
                        msg = msg + (' (mfn=%s)' % record[ISIS_MFN_KEY])
                    raise TypeError(msg % (id, id_tag, i))
                record['_id'] = id
                ids.add(id)
        elif gen_uuid:
            record['_id'] = unicode(uuid4())
        elif mfn:
            record['_id'] = record[ISIS_MFN_KEY]
        if prefix:
            # 迭代一个固定的标签序列
            for tag in tuple(record):
                if str(tag).isdigit():
                    record[prefix+tag] = record[tag]
                    del record[tag]  # 这就是迭代元组的原因
                    # 获取标签，但不直接从记录字典中获取
        if constant:
            constant_key, constant_value = constant.split(':')
            record[constant_key] = constant_value
        output.write(json.dumps(record).encode('utf-8'))
        output.write('\n')
    if not mongo:
        output.write(']\n')


def main(): ❹
    # 创建解析器
    parser = argparse.ArgumentParser(
        description='Convert an ISIS .mst or .iso file to a JSON array')
```

```
# 添加参数
parser.add_argument(
    'file_name', metavar='INPUT.(mst|iso)',
    help='.mst or .iso file to read')
parser.add_argument(
    '-o', '--out', type=argparse.FileType('w'), default=sys.stdout,
    metavar='OUTPUT.json',
    help='the file where the JSON output should be written'
         ' (default: write to stdout)')
parser.add_argument(
    '-c', '--couch', action='store_true',
    help='output array within a "docs" item in a JSON document'
         ' for bulk insert to CouchDB via POST to db/_bulk_docs')
parser.add_argument(
    '-m', '--mongo', action='store_true',
    help='output individual records as separate JSON dictionaries, one'
         ' per line for bulk insert to MongoDB via mongoimport utility')
parser.add_argument(
    '-t', '--type', type=int, metavar='ISIS_JSON_TYPE', default=1,
    help='ISIS-JSON type, sets field structure: 1=string, 2=alist,'
         ' 3=dict (default=1)')
parser.add_argument(
    '-q', '--qty', type=int, default=DEFAULT_QTY,
    help='maximum quantity of records to read (default=ALL)')
parser.add_argument(
    '-s', '--skip', type=int, default=0,
    help='records to skip from start of .mst (default=0)')
parser.add_argument(
    '-i', '--id', type=int, metavar='TAG_NUMBER', default=0,
    help='generate an "_id" from the given unique TAG field number'
         ' for each record')
parser.add_argument(
    '-u', '--uuid', action='store_true',
    help='generate an "_id" with a random UUID for each record')
parser.add_argument(
    '-p', '--prefix', type=str, metavar='PREFIX', default='',
    help='concatenate prefix to every numeric field tag'
         ' (ex. 99 becomes "v99")')
parser.add_argument(
    '-n', '--mfn', action='store_true',
    help='generate an "_id" from the MFN of each record'
         ' (available only for .mst input)')
parser.add_argument(
    '-k', '--constant', type=str, metavar='TAG:VALUE', default='',
    help='Include a constant tag:value in every record (ex. -k type:AS)')

'''
```

```
    # TODO: 实现这个功能，导出大量记录供给CouchDB
    parser.add_argument(
        '-r', '--repeat', type=int, default=1,
        help='repeat operation, saving multiple JSON files'
             ' (default=1, use -r 0 to repeat until end of input)')
    '''
    # 解析命令行
    args = parser.parse_args()
    if args.file_name.lower().endswith('.mst'):
        input_gen_func = iter_mst_records  ❺
    else:
        if args.mfn:
            print('UNSUPORTED: -n/--mfn option only available for .mst input.
            raise SystemExit
        input_gen_func = iter_iso_records  ❻
    input_gen = input_gen_func(args.file_name, args.type)  ❼
    if args.couch:
        args.out.write('{ "docs" : ')
    write_json(input_gen, args.file_name, args.out, args.qty,  ❽
               args.skip, args.id, args.uuid, args.mongo, args.mfn,
               args.type, args.prefix, args.constant)
    if args.couch:
        args.out.write('}\n')
    args.out.close()


if __name__ == '__main__':
    main()
```

❶ `iter_iso_records` 生成器函数读取 .iso 文件，产出记录。

❷ `iter_mst_records` 生成器函数读取 .mst 文件，产出记录。

❸ `write_json` 函数迭代 `input_gen` 生成器，输出 .json 文件。

❹ `main` 函数读取命令行参数，然后根据输入文件的扩展名选择……

❺ ……`iter_mst_records` 生成器函数……

❻ …… 或者 `iter_iso_records` 生成器函数。

❼ 使用选中的生成器函数构建生成器对象。

❽ 把生成器作为第一个参数传给 write_json 函数。

A.5 第16章：出租车队离散事件仿真

示例 A-6 是 16.9.2 节讨论的 taxi_sim.py 脚本的完整代码。

示例 **A-6** taxi_sim.py：出租车队仿真程序

```
"""
出租车仿真程序
==============

在控制台中驱动出租车::

    >>> from taxi_sim import taxi_process
    >>> taxi = taxi_process(ident=13, trips=2, start_time=0)
    >>> next(taxi)

Event(time=0, proc=13, action='leave garage')
>>> taxi.send(_.time + 7)
Event(time=7, proc=13, action='pick up passenger')
>>> taxi.send(_.time + 23)
Event(time=30, proc=13, action='drop off passenger')
>>> taxi.send(_.time + 5)
Event(time=35, proc=13, action='pick up passenger')
>>> taxi.send(_.time + 48)
Event(time=83, proc=13, action='drop off passenger')
>>> taxi.send(_.time + 1)
Event(time=84, proc=13, action='going home')
>>> taxi.send(_.time + 10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

运行示例：有两辆出租车，随机种子是10。这是有效的doctest::

```
>>> main(num_taxis=2, seed=10)
taxi: 0  Event(time=0, proc=0, action='leave garage')
taxi: 0  Event(time=5, proc=0, action='pick up passenger')
taxi: 1     Event(time=5, proc=1, action='leave garage')
taxi: 1     Event(time=10, proc=1, action='pick up passenger')
taxi: 1     Event(time=15, proc=1, action='drop off passenger')
taxi: 0  Event(time=17, proc=0, action='drop off passenger')
```

```
taxi: 1     Event(time=24, proc=1, action='pick up passenger')
taxi: 0  Event(time=26, proc=0, action='pick up passenger')
taxi: 0  Event(time=30, proc=0, action='drop off passenger')
taxi: 0  Event(time=34, proc=0, action='going home')
taxi: 1     Event(time=46, proc=1, action='drop off passenger')
taxi: 1     Event(time=48, proc=1, action='pick up passenger')
taxi: 1     Event(time=110, proc=1, action='drop off passenger')
taxi: 1     Event(time=139, proc=1, action='pick up passenger')
taxi: 1     Event(time=140, proc=1, action='drop off passenger')
taxi: 1     Event(time=150, proc=1, action='going home')
*** end of events ***
```

模块末尾有个更长的运行示例。

```
"""

import random
import collections
import queue
import argparse
import time

DEFAULT_NUMBER_OF_TAXIS = 3
DEFAULT_END_TIME = 180
SEARCH_DURATION = 5
TRIP_DURATION = 20
DEPARTURE_INTERVAL = 5

Event = collections.namedtuple('Event', 'time proc action')

# BEGIN TAXI_PROCESS
def taxi_process(ident, trips, start_time=0):
    """每次状态变化时向仿真程序产出一个事件"""
    time = yield Event(start_time, ident, 'leave garage')
    for i in range(trips):
        time = yield Event(time, ident, 'pick up passenger')
        time = yield Event(time, ident, 'drop off passenger')

    yield Event(time, ident, 'going home')
    # 结束出租车进程
# END TAXI_PROCESS


# BEGIN TAXI_SIMULATOR
class Simulator:
```

```
    def __init__(self, procs_map):
        self.events = queue.PriorityQueue()
        self.procs = dict(procs_map)

    def run(self, end_time):
        """调度并显示事件，直到时间结束"""
        # 调度各辆出租车的第一个事件
        for _, proc in sorted(self.procs.items()):
            first_event = next(proc)
            self.events.put(first_event)

        # 此次仿真的主循环
        sim_time = 0
        while sim_time < end_time:
            if self.events.empty():
                print('*** end of events ***')
                break

            current_event = self.events.get()
            sim_time, proc_id, previous_action = current_event
            print('taxi:', proc_id, proc_id * '   ', current_event)
            active_proc = self.procs[proc_id]
            next_time = sim_time + compute_duration(previous_action)
            try:
                next_event = active_proc.send(next_time)
            except StopIteration:
                del self.procs[proc_id]
            else:
                self.events.put(next_event)
        else:
            msg = '*** end of simulation time: {} events pending ***'
            print(msg.format(self.events.qsize()))
# END TAXI_SIMULATOR


def compute_duration(previous_action):
    """使用指数分布计算操作的耗时"""
    if previous_action in ['leave garage', 'drop off passenger']:
        # 新状态是四处徘徊
        interval = SEARCH_DURATION
    elif previous_action == 'pick up passenger':
        # 新状态是行程开始
        interval = TRIP_DURATION
    elif previous_action == 'going home':
        interval = 1
    else:
        raise ValueError('Unknown previous_action: %s' % previous_action)
```

```
    return int(random.expovariate(1/interval)) + 1


def main(end_time=DEFAULT_END_TIME, num_taxis=DEFAULT_NUMBER_OF_TAXIS,
         seed=None):
    """初始化随机生成器，构建过程，运行仿真程序"""
    if seed is not None:
        random.seed(seed)  # 获得可复现的结果

    taxis = {i: taxi_process(i, (i+1)*2, i*DEPARTURE_INTERVAL)
             for i in range(num_taxis)}
    sim = Simulator(taxis)
    sim.run(end_time)


if __name__ == '__main__':

    parser = argparse.ArgumentParser(
                        description='Taxi fleet simulator.')
    parser.add_argument('-e', '--end-time', type=int,
                        default=DEFAULT_END_TIME,
                        help='simulation end time; default = %s'
                        % DEFAULT_END_TIME)
    parser.add_argument('-t', '--taxis', type=int,
                        default=DEFAULT_NUMBER_OF_TAXIS,
                        help='number of taxis running; default = %s'
                        % DEFAULT_NUMBER_OF_TAXIS)
    parser.add_argument('-s', '--seed', type=int, default=None,
                        help='random generator seed (for testing)')

    args = parser.parse_args()
    main(args.end_time, args.taxis, args.seed)


"""

命令行中的运行示例：seed=3，最长用时=120::

# BEGIN TAXI_SAMPLE_RUN
$ python3 taxi_sim.py -s 3 -e 120
taxi: 0  Event(time=0, proc=0, action='leave garage')
taxi: 0  Event(time=2, proc=0, action='pick up passenger')
taxi: 1     Event(time=5, proc=1, action='leave garage')
taxi: 1     Event(time=8, proc=1, action='pick up passenger')
taxi: 2        Event(time=10, proc=2, action='leave garage')
taxi: 2        Event(time=15, proc=2, action='pick up passenger')
taxi: 2        Event(time=17, proc=2, action='drop off passenger')
```

```
taxi: 0  Event(time=18, proc=0, action='drop off passenger')
taxi: 2        Event(time=18, proc=2, action='pick up passenger')
taxi: 2        Event(time=25, proc=2, action='drop off passenger')
taxi: 1     Event(time=27, proc=1, action='drop off passenger')
taxi: 2        Event(time=27, proc=2, action='pick up passenger')
taxi: 0  Event(time=28, proc=0, action='pick up passenger')
taxi: 2        Event(time=40, proc=2, action='drop off passenger')
taxi: 2        Event(time=44, proc=2, action='pick up passenger')
taxi: 1     Event(time=55, proc=1, action='pick up passenger')
taxi: 1     Event(time=59, proc=1, action='drop off passenger')
taxi: 0  Event(time=65, proc=0, action='drop off passenger')
taxi: 1     Event(time=65, proc=1, action='pick up passenger')
taxi: 2        Event(time=65, proc=2, action='drop off passenger')
taxi: 2        Event(time=72, proc=2, action='pick up passenger')
taxi: 0  Event(time=76, proc=0, action='going home')
taxi: 1     Event(time=80, proc=1, action='drop off passenger')
taxi: 1     Event(time=88, proc=1, action='pick up passenger')
taxi: 2        Event(time=95, proc=2, action='drop off passenger')
taxi: 2        Event(time=97, proc=2, action='pick up passenger')
taxi: 2        Event(time=98, proc=2, action='drop off passenger')
taxi: 1     Event(time=106, proc=1, action='drop off passenger')
taxi: 2        Event(time=109, proc=2, action='going home')
taxi: 1     Event(time=110, proc=1, action='going home')
*** end of events ***
# END TAXI_SAMPLE_RUN

"""
```

A.6　第17章：加密示例

这几个脚本用于展示如何使用 `futures.ProcessPoolExecutor` 执行 CPU 密集型任务。

示例 A-7 使用 RC4 算法加密并解密随机的字节数组，需要 arcfour.py 模块（见示例 A-8）支持才能运行。

示例 **A-7**　arcfour_futures.py：`futures.ProcessPoolExecutor` 用法示例

```
import sys
import time
from concurrent import futures
from random import randrange
from arcfour import arcfour

JOBS = 12
SIZE = 2**18

KEY = b"'Twas brillig, and the slithy toves\nDid gyre"
STATUS = '{} workers, elapsed time: {:.2f}s'


def arcfour_test(size, key):
    in_text = bytearray(randrange(256) for i in range(size))
    cypher_text = arcfour(key, in_text)
    out_text = arcfour(key, cypher_text)
    assert in_text == out_text, 'Failed arcfour_test'
    return size


def main(workers=None):
    if workers:
        workers = int(workers)
    t0 = time.time()

    with futures.ProcessPoolExecutor(workers) as executor:
        actual_workers = executor._max_workers
        to_do = []
        for i in range(JOBS, 0, -1):
            size = SIZE + int(SIZE / JOBS * (i - JOBS/2))
```

```
            job = executor.submit(arcfour_test, size, KEY)
            to_do.append(job)

        for future in futures.as_completed(to_do):
            res = future.result()
            print('{:.1f} KB'.format(res/2**10))

    print(STATUS.format(actual_workers, time.time() - t0))

if __name__ == '__main__':
    if len(sys.argv) == 2:
        workers = int(sys.argv[1])
    else:
        workers = None
    main(workers)
```

示例 A-8 纯粹使用 Python 实现 RC4 加密算法。

示例 **A-8**　arcfour.py：兼容 RC4 的算法

```
"""兼容RC4的算法"""

def arcfour(key, in_bytes, loops=20):

    kbox = bytearray(256)  # 创建存储键的数组
    for i, car in enumerate(key):  # 复制键和向量
        kbox[i] = car
    j = len(key)
    for i in range(j, 256):  # 重复到底
        kbox[i] = kbox[i-j]

    # [1] 初始化sbox
    sbox = bytearray(range(256))

    # 按照CipherSaber-2的建议，不断打乱sbox
    # http://ciphersaber.gurus.com/faq.html#cs2
    j = 0
    for k in range(loops):
        for i in range(256):
            j = (j + sbox[i] + kbox[i]) % 256
            sbox[i], sbox[j] = sbox[j], sbox[i]

    # 主循环
    i = 0
    j = 0
```

```
    out_bytes = bytearray()

    for car in in_bytes:
        i = (i + 1) % 256
        # [2] 打乱sbox
        j = (j + sbox[i]) % 256
        sbox[i], sbox[j] = sbox[j], sbox[i]
        # [3] 计算t
        t = (sbox[i] + sbox[j]) % 256
        k = sbox[t]
        car = car ^ k
        out_bytes.append(car)

    return out_bytes


def test():
    from time import time
    clear = bytearray(b'1234567890' * 100000)
    t0 = time()
    cipher = arcfour(b'key', clear)
    print('elapsed time: %.2fs' % (time() - t0))
    result = arcfour(b'key', cipher)
    assert result == clear, '%r != %r' % (result, clear)
    print('elapsed time: %.2fs' % (time() - t0))
    print('OK')


if __name__ == '__main__':
    test()
```

示例 A-9 使用 SHA-256 散列算法打乱字节数组。这个脚本使用标准库中的 hashlib 模块，而这个模块使用 C 语言编写的 OpenSSL 库。

示例 **A-9**　sha_futures.py：`futures.ProcessPoolExecutor` 用法示例

```
import sys
import time
import hashlib
from concurrent import futures
from random import randrange

JOBS = 12
```

```
SIZE = 2**20
STATUS = '{} workers, elapsed time: {:.2f}s'


def sha(size):
    data = bytearray(randrange(256) for i in range(size))
    algo = hashlib.new('sha256')
    algo.update(data)
    return algo.hexdigest()


def main(workers=None):
    if workers:
        workers = int(workers)
    t0 = time.time()

    with futures.ProcessPoolExecutor(workers) as executor:
        actual_workers = executor._max_workers
        to_do = (executor.submit(sha, SIZE) for i in range(JOBS))
        for future in futures.as_completed(to_do):
            res = future.result()
            print(res)

    print(STATUS.format(actual_workers, time.time() - t0))

if __name__ == '__main__':
    if len(sys.argv) == 2:
        workers = int(sys.argv[1])
    else:
        workers = None
    main(workers)
```

A.7　第17章：flags2系列HTTP客户端示例

17.5 节的 flags2 系列示例都使用了 flags2_common.py 模块（见示例 A-10）里的函数。

示例 **A-10**　flags2_common.py

```
"""为后续flag示例提供实用函数。
"""

import os
import time
import sys
import string
import argparse
from collections import namedtuple
from enum import Enum


Result = namedtuple('Result', 'status data')

HTTPStatus = Enum('Status', 'ok not_found error')

POP20_CC = ('CN IN US ID BR PK NG BD RU JP '
            'MX PH VN ET EG DE IR TR CD FR').split()

DEFAULT_CONCUR_REQ = 1
MAX_CONCUR_REQ = 1

SERVERS = {
    'REMOTE': 'http://flupy.org/data/flags',
    'LOCAL':  'http://localhost:8001/flags',
    'DELAY':  'http://localhost:8002/flags',
    'ERROR':  'http://localhost:8003/flags',
}
DEFAULT_SERVER = 'LOCAL'

DEST_DIR = 'downloads/'
COUNTRY_CODES_FILE = 'country_codes.txt'
```

```
def save_flag(img, filename):
    path = os.path.join(DEST_DIR, filename)
    with open(path, 'wb') as fp:
        fp.write(img)


def initial_report(cc_list, actual_req, server_label):
    if len(cc_list) <= 10:
        cc_msg = ', '.join(cc_list)
    else:
        cc_msg = 'from {} to {}'.format(cc_list[0], cc_list[-1])
    print('{} site: {}'.format(server_label, SERVERS[server_label]))
    msg = 'Searching for {} flag{}: {}'
    plural = 's' if len(cc_list) != 1 else ''
    print(msg.format(len(cc_list), plural, cc_msg))
    plural = 's' if actual_req != 1 else ''
    msg = '{} concurrent connection{} will be used.'
    print(msg.format(actual_req, plural))


def final_report(cc_list, counter, start_time):
    elapsed = time.time() - start_time
    print('-' * 20)
    msg = '{} flag{} downloaded.'
    plural = 's' if counter[HTTPStatus.ok] != 1 else ''
    print(msg.format(counter[HTTPStatus.ok], plural))
    if counter[HTTPStatus.not_found]:
        print(counter[HTTPStatus.not_found], 'not found.')
    if counter[HTTPStatus.error]:
        plural = 's' if counter[HTTPStatus.error] != 1 else ''
        print('{} error{}.'.format(counter[HTTPStatus.error], plural))
    print('Elapsed time: {:.2f}s'.format(elapsed))


def expand_cc_args(every_cc, all_cc, cc_args, limit):
    codes = set()
    A_Z = string.ascii_uppercase
    if every_cc:
        codes.update(a+b for a in A_Z for b in A_Z)
    elif all_cc:
        with open(COUNTRY_CODES_FILE) as fp:
            text = fp.read()
        codes.update(text.split())
    else:
        for cc in (c.upper() for c in cc_args):
            if len(cc) == 1 and cc in A_Z:
                codes.update(cc+c for c in A_Z)
```

```
            elif len(cc) == 2 and all(c in A_Z for c in cc):
                codes.add(cc)
            else:
                msg = 'each CC argument must be A to Z or AA to ZZ.'
                raise ValueError('*** Usage error: '+msg)
    return sorted(codes)[:limit]


def process_args(default_concur_req):
    server_options = ', '.join(sorted(SERVERS))
    parser = argparse.ArgumentParser(
                description='Download flags for country codes. '
                'Default: top 20 countries by population.')
    parser.add_argument('cc', metavar='CC', nargs='*',
                help='country code or 1st letter (eg. B for BA...BZ)')
    parser.add_argument('-a', '--all', action='store_true',
                help='get all available flags (AD to ZW)')
    parser.add_argument('-e', '--every', action='store_true',
                help='get flags for every possible code (AA...ZZ)')
    parser.add_argument('-l', '--limit', metavar='N', type=int,
                help='limit to N first codes', default=sys.maxsize)
    parser.add_argument('-m', '--max_req', metavar='CONCURRENT', type=int,
                default=default_concur_req,
                help='maximum concurrent requests (default={})'
                      .format(default_concur_req))
    parser.add_argument('-s', '--server', metavar='LABEL',
                default=DEFAULT_SERVER,
                help='Server to hit; one of {} (default={})'
                      .format(server_options, DEFAULT_SERVER))
    parser.add_argument('-v', '--verbose', action='store_true',
                help='output detailed progress info')
    args = parser.parse_args()
    if args.max_req < 1:
        print('*** Usage error: --max_req CONCURRENT must be >= 1')
        parser.print_usage()
        sys.exit(1)
    if args.limit < 1:
        print('*** Usage error: --limit N must be >= 1')
        parser.print_usage()
        sys.exit(1)
    args.server = args.server.upper()
    if args.server not in SERVERS:
        print('*** Usage error: --server LABEL must be one of',
              server_options)
        parser.print_usage()
        sys.exit(1)
    try:
```

```
        cc_list = expand_cc_args(args.every, args.all, args.cc, args.limit)
    except ValueError as exc:
        print(exc.args[0])
        parser.print_usage()
        sys.exit(1)
    if not cc_list:
        cc_list = sorted(POP20_CC)
    return args, cc_list

def main(download_many, default_concur_req, max_concur_req):
    args, cc_list = process_args(default_concur_req)
    actual_req = min(args.max_req, max_concur_req, len(cc_list))
    initial_report(cc_list, actual_req, args.server)
    base_url = SERVERS[args.server]
    t0 = time.time()
    counter = download_many(cc_list, base_url, args.verbose, actual_req)
    assert sum(counter.values()) == len(cc_list), \
        'some downloads are unaccounted for'
    final_report(cc_list, counter, t0)
```

flags2_sequential.py 脚本（见示例 A-11）是对比两种并发实现的基准。flags2_threadpool.py 脚本（见示例 17-14）还使用了 flags2_sequential.py 脚本中的 `get_flag` 和 `download_one` 两个函数。

示例 **A-11**　flags2_sequential.py

```
"""下载多个国家的国旗（包含错误处理代码）。

依序下载版

运行示例::

    $ python3 flags2_sequential.py -s DELAY b
    DELAY site: http://localhost:8002/flags
    Searching for 26 flags: from BA to BZ
    1 concurrent connection will be used.
    --------------------
    17 flags downloaded.
    9 not found.
    Elapsed time: 13.36s

"""

import collections
```

```
import requests
import tqdm

from flags2_common import main, save_flag, HTTPStatus, Result


DEFAULT_CONCUR_REQ = 1
MAX_CONCUR_REQ = 1

# BEGIN FLAGS2_BASIC_HTTP_FUNCTIONS
def get_flag(base_url, cc):
    url = '{}/{cc}/{cc}.gif'.format(base_url, cc=cc.lower())
    resp = requests.get(url)

    if resp.status_code != 200:
        resp.raise_for_status()
    return resp.content

def download_one(cc, base_url, verbose=False):
    try:
        image = get_flag(base_url, cc)
    except requests.exceptions.HTTPError as exc:
        res = exc.response
        if res.status_code == 404:
            status = HTTPStatus.not_found
            msg = 'not found'
        else:
            raise
    else:
        save_flag(image, cc.lower() + '.gif')
        status = HTTPStatus.ok
        msg = 'OK'

    if verbose:
        print(cc, msg)

    return Result(status, cc)
# END FLAGS2_BASIC_HTTP_FUNCTIONS

# BEGIN FLAGS2_DOWNLOAD_MANY_SEQUENTIAL
def download_many(cc_list, base_url, verbose, max_req):
    counter = collections.Counter()
    cc_iter = sorted(cc_list)
    if not verbose:
        cc_iter = tqdm.tqdm(cc_iter)
    for cc in cc_iter:
```

```
        try:
            res = download_one(cc, base_url, verbose)
        except requests.exceptions.HTTPError as exc:
            error_msg = 'HTTP error {res.status_code} - {res.reason}'
            error_msg = error_msg.format(res=exc.response)
        except requests.exceptions.ConnectionError as exc:
            error_msg = 'Connection error'
        else:
            error_msg = ''
            status = res.status

        if error_msg:
            status = HTTPStatus.error
        counter[status] += 1
        if verbose and error_msg:
            print('*** Error for {}: {}'.format(cc, error_msg))

    return counter
# END FLAGS2_DOWNLOAD_MANY_SEQUENTIAL

if __name__ == '__main__':
    main(download_many, DEFAULT_CONCUR_REQ, MAX_CONCUR_REQ)
```

A.8　第19章：处理OSCON日程表的脚本和测试

示例 A-12 是 schedule1.py 模块（示例 19-9）的测试脚本，使用 `py.test` 库和测试运行程序实现。

示例 **A-12**　test_schedule1.py

```
import shelve
import pytest

import schedule1 as schedule


@pytest.yield_fixture
def db():
    with shelve.open(schedule.DB_NAME) as the_db:
        if schedule.CONFERENCE not in the_db:
            schedule.load_db(the_db)
        yield the_db


def test_record_class():
    rec = schedule.Record(spam=99, eggs=12)
    assert rec.spam == 99
    assert rec.eggs == 12


def test_conference_record(db):
    assert schedule.CONFERENCE in db


def test_speaker_record(db):
    speaker = db['speaker.3471']
    assert speaker.name == 'Anna Martelli Ravenscroft'


def test_event_record(db):
    event = db['event.33950']
    assert event.name == 'There *Will* Be Bugs'
```

```
def test_event_venue(db):
    event = db['event.33950']
    assert event.venue_serial == 1449
```

19.1.5 节分四部分列出了 schedule2.py 脚本里的代码，示例 A-13 是完整的代码清单。

示例 **A-13**　schedule2.py

```
"""
schedule2.py: 遍历OSCON的日程数据

    >>> import shelve
    >>> db = shelve.open(DB_NAME)
    >>> if CONFERENCE not in db: load_db(db)

# BEGIN SCHEDULE2_DEMO

    >>> DbRecord.set_db(db)
    >>> event = DbRecord.fetch('event.33950')
    >>> event
    <Event 'There *Will* Be Bugs'>
    >>> event.venue
    <DbRecord serial='venue.1449'>
    >>> event.venue.name
    'Portland 251'
    >>> for spkr in event.speakers:
    ...     print('{0.serial}: {0.name}'.format(spkr))
    ...
    speaker.3471: Anna Martelli Ravenscroft
    speaker.5199: Alex Martelli

# END SCHEDULE2_DEMO

    >>> db.close()

"""

# BEGIN SCHEDULE2_RECORD
import warnings
import inspect

import osconfeed

DB_NAME = 'data/schedule2_db'
```

```
CONFERENCE = 'conference.115'


class Record:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

    def __eq__(self, other):
        if isinstance(other, Record):
            return self.__dict__ == other.__dict__
        else:
            return NotImplemented
# END SCHEDULE2_RECORD


# BEGIN SCHEDULE2_DBRECORD
class MissingDatabaseError(RuntimeError):
    """Raised when a database is required but was not set."""


class DbRecord(Record):

    __db = None

    @staticmethod
    def set_db(db):
        DbRecord.__db = db

    @staticmethod
    def get_db():
        return DbRecord.__db

    @classmethod
    def fetch(cls, ident):
        db = cls.get_db()
        try:
            return db[ident]
        except TypeError:
            if db is None:
                msg = "database not set; call '{}.set_db(my_db)'"
                raise MissingDatabaseError(msg.format(cls.__name__))
            else:
                raise

    def __repr__(self):
        if hasattr(self, 'serial'):
            cls_name = self.__class__.__name__
```

```
            return '<{} serial={!r}>'.format(cls_name, self.serial)
        else:
            return super().__repr__()
# END SCHEDULE2_DBRECORD


# BEGIN SCHEDULE2_EVENT
class Event(DbRecord):

    @property
    def venue(self):
        key = 'venue.{}'.format(self.venue_serial)
        return self.__class__.fetch(key)

    @property
    def speakers(self):
        if not hasattr(self, '_speaker_objs'):
            spkr_serials = self.__dict__['speakers']
            fetch = self.__class__.fetch
            self._speaker_objs = [fetch('speaker.{}'.format(key))
                                  for key in spkr_serials]
        return self._speaker_objs

    def __repr__(self):
        if hasattr(self, 'name'):
            cls_name = self.__class__.__name__
            return '<{} {!r}>'.format(cls_name, self.name)
        else:
            return super().__repr__()
# END SCHEDULE2_EVENT


# BEGIN SCHEDULE2_LOAD
def load_db(db):
    raw_data = osconfeed.load()
    warnings.warn('loading ' + DB_NAME)
    for collection, rec_list in raw_data['Schedule'].items():
        record_type = collection[:-1]
        cls_name = record_type.capitalize()
        cls = globals().get(cls_name, DbRecord)
        if inspect.isclass(cls) and issubclass(cls, DbRecord):
            factory = cls
        else:
            factory = DbRecord
        for record in rec_list:
            key = '{}.{}'.format(record_type, record['serial'])
            record['serial'] = key
```

```
            db[key] = factory(**record)
# END SCHEDULE2_LOAD
```

示例 A-14 使用 `py.test` 测试示例 A-13。

示例 **A-14**　test_schedule2.py

```
import shelve
import pytest

import schedule2 as schedule


@pytest.yield_fixture
def db():
    with shelve.open(schedule.DB_NAME) as the_db:
        if schedule.CONFERENCE not in the_db:
            schedule.load_db(the_db)
        yield the_db


def test_record_attr_access():
    rec = schedule.Record(spam=99, eggs=12)
    assert rec.spam == 99
    assert rec.eggs == 12


def test_record_repr():
    rec = schedule.DbRecord(spam=99, eggs=12)
    assert 'DbRecord object at 0x' in repr(rec)
    rec2 = schedule.DbRecord(serial=13)
    assert repr(rec2) == "<DbRecord serial=13>"


def test_conference_record(db):
    assert schedule.CONFERENCE in db


def test_speaker_record(db):
    speaker = db['speaker.3471']
    assert speaker.name == 'Anna Martelli Ravenscroft'


def test_missing_db_exception():
    with pytest.raises(schedule.MissingDatabaseError):
```

```
        schedule.DbRecord.fetch('venue.1585')


def test_dbrecord(db):
    schedule.DbRecord.set_db(db)
    venue = schedule.DbRecord.fetch('venue.1585')
    assert venue.name == 'Exhibit Hall B'


def test_event_record(db):
    event = db['event.33950']
    assert repr(event) == "<Event 'There *Will* Be Bugs'>"


def test_event_venue(db):
    schedule.Event.set_db(db)
    event = db['event.33950']
    assert event.venue_serial == 1449
    assert event.venue == db['venue.1449']
    assert event.venue.name == 'Portland 251'


def test_event_speakers(db):
    schedule.Event.set_db(db)
    event = db['event.33950']
    assert len(event.speakers) == 2
    anna_and_alex = [db['speaker.3471'], db['speaker.5199']]
    assert event.speakers == anna_and_alex


def test_event_no_speakers(db):
    schedule.Event.set_db(db)
    event = db['event.36848']
    assert len(event.speakers) == 0
```

Python 术语表

当然，这里列出的很多术语不是 Python 专用的，不过某些术语的定义对 Python 社区有特殊的意义。

此外，也可以参阅官方的 Python 词汇表（https://docs.python.org/3/glossary.html）。

ABC（编程语言）

Leo Geurts、Lambert Meertens 和 Steven Pemberton 创造的一门编程语言。20 世纪 80 年代，Python 之父 Guido van Rossum 是实现 ABC 环境的程序员。Python 的一些特色出自 ABC，例如使用缩进划分块、内置元组和字典、元组拆包、`for` 循环的语义，以及对所有序列类型的统一处理方式。

BDFL

Benevolent Dictator For Life 的简称，意为“仁慈的独裁者”，指代 Python 之父 Guido van Rossum。

BOM

Byte Order Mark 的简称，意为“字节序标记”，指可能出现在 UTF-16 编码文件开头的字节序列。BOM 是 U+FEFF 字符（零宽不换行空格），在大字节序的 CPU 中，编码成 `b'\xfe\xff'`；在小字节序的 CPU 中，编码成 `b'\xff\xfe'`。因为 Unicode 中没有 U+FFFE 字符，所以这些字节的作用只有一个——表示编码方式使用的字节序。虽然多余，但是在 UTF-8 文件中可能会找到编码成 `b'\xef\xbb\xbf'` 的 BOM。

CPython

标准的 Python 解释器，使用 C 语言实现。讨论不同实现特有的行为，以及多个可用的 Python 解释器（如 PyPy）时才会使用这个术语。

CRUD

Create、Read、Update、Delete 的首字母缩写，这是存储记录的应用程序中的四种基本操作。

doctest

一个模块，其中的函数能解析并运行 Python 模块或纯文本文件的文档字符串中内嵌的示例。也可以在命令行中使用，如下所示：

```
python -m doctest
module_with_tests.py
```

DRY

Don't Repeat Yourself（不要自我重复）的缩写，一种软件工程原则，意思是："系统中的每一项知识都必须具有单一、无歧义、权威的表示。"首先由 Andy Hunt 与 Dave Thomas 的《程序员修炼之道：从小工到专家》一书提出。

dunder

首尾有两条下划线的特殊方法和属性的简洁读法（即把 `__len__` 读成"dunder len"）。

dunder 方法

参见 dunder 和特殊方法词条。

EAFP

"it's easier to ask forgiveness than permission"（取得原谅比获得许可容易）的首字母缩写。人们认为这句话是计算机先驱 Grace Hopper 说的，Python 程序员使用这个缩写指代一种动态编程方式，例如访问属性前不测试有没有属性，如果没有就捕获异常。 `hasattr` 函数的文档字符串是这样描述它的工作方式的："调用 `getattr(object, name)`，然后捕获 `AttributeError` 异常。"

genexp

generator expression（生成器表达式）的简称。

GoF 书

指代《设计模式：可复用面向对象软件的基础》一书，作者是四个人，被称为“四人组”（Gang of Four，GoF），包括 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides。

KISS 原则

KISS 是“Keep It Simple, Stupid”的首字母缩写。这个原则要求尽量寻找最简单的方案，尽量减少可变部分。这个警句是 Kelly Johnson 首创的。Kelly 是一位多才多艺的航空工程师，在真实存在的 51 区工作，设计出了 20 世纪最先进的几架航天飞机。

listcomp

list comprehension（列表推导）的简称。

ORM

Object-Relational Mapper（对象关系映射器）的缩写，通过这种 API 可以使用 Python 类和对象访问数据库中的表和记录，而且调用方法可以执行数据库操作。SQLAlchemy 是流行的独立 Python ORM，Django 和 Web2py 自带了 ORM。

PyPI

Python 包索引（https://pypi.python.org/），里面有超过 60 000 个包可用。也叫奶酪店（参见奶酪店词条）。为了防止与 PyPy 混淆，PyPI 应该读作“pie-P-eye”。

PyPy

Python 编程语言的另一种实现，使用一个工具链把部分 Python 编译成机器码，因此解释器的源码其实是使用 Python 编写的。PyPy 还提供

了 JIT，即时把用户的程序编译成机器码——与 Java VM 的作用相同。根据 PyPy 公布的基准测试（http://speed.pypy.org/），从 2014 年 11 月起，PyPy 平均比 CPython 快 6.8 倍。为了防止与 PyPI 混淆， PyPy 应该读作“pie-pie”。

Pythonic

用于赞扬符合 Python 风格的代码，即充分利用 Python 语言的特性，写出简洁明了、可读性强，通常运行速度也快的代码。还指 API 符合 Python 高手的编程方式。参见惯用句法词条。

Python 之禅

从 Python 2.2 起，在 Python 控制台中输入 `import this` 后看到的输出。

REPL

read-eval-print loop（读取－求值－输出循环）的简称，一种交互式控制台，如标准的 `python` 或非主流的 `ipython` 和 `bpython`，以及 Python Anywhere。

YAGNI

You Ain't Gonna Need It（你不需要这个）的首字母缩写，这个口号的意思是，根据对未来需求的预测，不要实现非立即需要的功能。

绑定方法（**bound method**）

通过实例访问的方法会绑定到那个实例上。方法其实是描述符，访问方法时，会返回一个包装自身的对象，把方法绑定到实例上。那个对象就是绑定方法。调用绑定方法时，可以不传入 `self` 的值。例如，像 `my_method = my_obj.method` 这样赋值之后，可以通过 `my_method()` 调用绑定方法。请与非绑定方法相比较。

编码解码器（**codec**）

（编码器 / 解码器）提供编码和解码函数的模块，通常在 `str` 和

bytes 之间转换，不过 Python 也提供了在 bytes 和 bytes，以及 str 和 str 之间转换的编码解码器。

变值方法（**mutator**）

参见存取方法词条。

别名（**aliasing**）

为同一个对象指定两个或多个名称。例如，在 `a = []; b = a` 中，a 和 b 是别名，指向同一个列表对象。对于把对象引用存储在变量中的语言来说，别名无处不在。为了避免混淆，要摒弃这种想法：变量是存储对象的盒子（毕竟同一个对象不可能放在两个盒子里）。我们要把变量看做对象的标注（一个对象可以有多个标注）。

并行赋值（**parallel assignment**）

使用类似 `a, b = [c, d]` 这样的句法，把可迭代对象中的元素赋值给多个变量，也叫解构赋值。这是元组拆包的常见用途。

抽象基类（**abstract base class**，**ABC**）

无法实例化，只能扩展的类。Python 通过 ABC 实现接口。除了继承 ABC 之外，类还可以注册成为 ABC 的虚拟子类，声明自己实现了接口。

初始化方法（**initializer**）

`__init__` 方法更贴切的名称（取代构造方法）。`__init__` 方法的任务是初始化通过 self 参数传入的实例。实例其实是由 `__new__` 方法构建的。参见构造方法词条。

储存属性（**storage attribute**）

托管实例中的属性，用于存储由描述符管理的属性的值。另见托管属性词条。

存取方法（**accessor**）

用于存取单个数据属性的方法。有些作者把存取方法当作通用术语使用，包括读值方法和设值方法；另一些作者则用存取方法指代读值方法，而用变值方法指代设值方法。

代码异味（**code smell**）

一种代码形式，表明程序的设计可能有问题。例如，过度使用 `isinstance` 检查具体的类是一种代码异味，因为这样会导致程序以后难以扩展，无法处理新类型。

单例（**singleton**）

一个类唯一存在的实例——这通常不是巧合，而是故意为之，防止类创建多个实例。有一种设计模式就叫单例模式，指明如何编写这样的类。在 Python 中，`None` 对象是单例。

导入时（**import time**）

Python 解释器加载模块，从上到下计算，把里面的代码编译成字节码之后，开始执行模块的那一刻。类和函数在此时定义，变成真实存在的对象。装饰器也在此时执行。

迭代器（**iterator**）

实现了无参数方法 `__next__` 的对象；这个方法返回级数里的下一个元素，如果没有元素了就抛出 `StopIteration` 异常。在 Python 中，迭代器还实现了 `__iter__` 方法，因此迭代器也是可迭代的对象。根据最初的设计模式，经典迭代器返回集合里的元素。生成器也是迭代器，不过更灵活。参见生成器词条。

惰性求值（**lazy**）

指可迭代的对象按需生成元素。在 Python 中，生成器会惰性求值。请与及早求值相比较。

二进制序列（**binary sequence**）

一个通用术语，表示元素是二进制数据的序列类型。内置的二进制

序列类型有 `byte`、`bytearray` 和 `memoryview`。

泛函数（**generic function**）

以不同的方式为不同类型的对象实现相同操作的一组函数。从 Python 3.4 起，创建泛函数的标准方式是使用 `functools.singledispatch` 装饰器。在其他语言中，这叫多分派方法。

非绑定方法（**unbound method**）

直接通过类访问的实例方法没有绑定到特定的实例上，因此把这种方法称为“非绑定方法”。若想成功调用非绑定方法，必须显式传入类的实例作为第一个参数。那个实例会赋值给方法的 `self` 参数。参见绑定方法词条。

非覆盖型描述符（**nonoverriding descriptor**）

未实现 `__set__` 方法的描述符，不干涉托管实例中托管属性的设置。因此，托管实例中的同名属性会遮盖实例中的描述符。也叫非数据描述符或遮盖型描述符。请与覆盖型描述符相比较。

覆盖型描述符（**overriding descriptor**）

实现了 `__set__` 方法的描述符，设置托管实例中的托管属性时会遭到拦截并覆盖相关操作。也叫数据描述符或强制描述符。请与非覆盖型描述符相比较。

高阶函数（**higher-order function**）

以其他函数为参数的函数，例如 `sorted`、`map` 和 `filter`；或者，返回值为函数的函数，例如 Python 中的装饰器。

构造方法（**constructor**）

类的 `__init__` 实例方法称为类的构造方法，因为这个方法的语义类似于 Java 中的构造方法。然而，这样称呼并不规范，`__init__` 更应该称为初始化方法，因为它并不会构建实例，而是把实例传给 `self` 参

数。Python 在 `__init__` 方法之前调用的 `__new__` 类方法更合乎构造方法这个术语，`__new__` 方法才会创建实例并将其返回。参见初始化方法词条。

惯用句法（**idiom**）

根据普林斯顿大学 WordNet 字典的定义，惯用句法指“说母语的人说话的方式”。

函数（**function**）

严格来说，是指 `def` 块或 `lambda` 表达式计算得到的对象。通常，函数这个词用于表示任何可调用的对象，例如方法，有时甚至表示类。官方文档中的内置函数列表（http://docs.python.org/library/functions.html）列出了几个内置的类，例如 `dict`、`range` 和 `str`。另见可调用的对象词条。

猴子补丁（**monkey patching**）

在运行时动态修改模块、类或函数，通常是添加功能或修正缺陷。猴子补丁在内存中发挥作用，不会修改源码，因此只对当前运行的程序实例有效。因为猴子补丁破坏了封装，而且容易导致程序与补丁代码的实现细节紧密耦合，所以被视为临时的变通方案，不是集成代码的推荐方式。

混入方法（**mixin method**）

抽象基类或混入类中方法的具体实现。

混入类（**mixin class**）

用于随着多重继承类树中的一个或多个类一起扩展的类。混入类绝不能实例化，它的具体子类也应该是其他非混入类的子类。

活性（**liveness**）

异步系统、线程系统或分布式系统在“期待的事情终于发生”（即虽然期待的计算不会立即发生，但最终会完成）时展现出来的特性叫活

性。如果系统死锁了，活性也就没有了。

及早求值（**eager**）

指可迭代对象一次构建好全部元素。在 Python 中，列表推导会及早求值。请与惰性求值相比较。

集合（**collection**）

泛指由元素组成，可以单独访问各个元素的数据结构。有些集合可以包含任意类型的对象（参见容器词条），有些则只能包含一种原子类型的对象（参见平坦序列词条）。`list` 和 `bytes` 都是集合，只不过 `list` 是容器，而 `bytes` 是平坦序列。

假值（**falsy**）

只要 `bool(x)` 返回 `False`，`x` 就是假值。需要布尔值时，Python 会隐式使用 `bool` 计算对象，例如控制 `if` 和 `while` 循环的表达式。与此相对的是真值（truthy）。

尽早失败（**fail-fast**）

一种系统设计方式，建议应该尽早报告错误。Python 比其他大多数动态编程语言更遵守这一原则。例如，Python 中没有“未定义”的值：在初始化之前引用变量会报错；如果 `k` 不存在，`my_dict[k]` 会抛出异常（JavaScript 则不然）。还有一例：在 Python 中通过元组拆包做并行赋值，必须显式处理元组的每一个元素才行；而在 Ruby 中，如果 `=` 两边的元素数量不一致，右边未用到的元素会被忽略，或者把 `nil` 赋给左边多余的变量。

可迭代的（**iterable**）

使用内置的 `iter` 函数可以从中获得迭代器的对象。可迭代的对象为 `for` 循环、列表推导和元组拆包提供元素。如果对象的 `__iter__` 方法能返回迭代器，这就是可迭代的对象。序列都是可迭代的对象；此外，实现 `__getitem__` 方法的对象也是可迭代的对象。

可迭代对象的拆包（**iterable unpacking**）

元组拆包更现代、更精确的同义词。另见并行赋值词条。

可散列的（**hashable**）

在散列值永不改变，而且如果 `a == b`，那么 `hash(a) == hash(b)` 也是 `True` 的情况下，如果对象既有 `__hash__` 方法，也有 `__eq__` 方法，那么这样的对象称为可散列的对象。在内置的类型中，大多数不可变的类型都是可散列的；但是，仅当元组的每一个元素都是可散列的时，元组才是可散列的。

可调用的对象（**callable object**）

可以使用调用运算符 `()` 调用，能返回结果或执行某项操作的对象。在 Python 中，可调用的对象有七种：用户定义的函数、内置的函数、内置的方法、实例方法、生成器函数、类，还有实现特殊方法 `__call__` 的类的实例。

类（**class**）

定义新类型的程序结构，里面有数据属性，以及用于操作数据属性的方法。参见类型词条。

类型（**type**）

程序中的各种数据，限定可取的值和可对数据做的操作。有些 Python 类型近似于机器数据类型（例如 `float` 和 `bytes`），而另一些则是机器数据类型的扩展（例如，`int` 不受 CPU 字长的限制，`str` 包含多字节 Unicode 数据码位）和特别高层的抽象（例如 `dict`、 `deque`，等等）。类型分为两类：用户定义的类型和解释器内置的类型。在 Python 2.2 统一类型和类之前，类型和类是不同的实体，用户定义的类不能扩展内置的类型。而在那之后，内置的类型和新式类兼容了，类是 `type` 的实例。在 Python 3 中，所有类都是新式类。参见类和元类词条。

列表推导（**list comprehension**）

放在方括号里的表达式，使用关键字 `for` 和 `in`，通过处理和过滤一个或多个可迭代对象里的元素构建列表。列表推导会及早求值。参见及早求值词条。

码位（**code point**）

介于 0~0x10FFFF 之间的整数，用于标识 Unicode 字符数据库中的字符。截至 Unicode 7.0，所有码位中只有不到 3% 指定了字符。在 Python 文档中，这个术语可能拼成一个词，也可能拼成两个词。例如，在 Python 标准库参考手册的“2. Built-in Functions”一章（http://docs.python.org/library/functions.html）中，说 `char` 函数的参数是一个整数“码位”（codepoint），却说作用相反的 `ord` 函数返回一个“Unicode 码位”（Unicode code point）。

描述符（**descriptor**）

一个类，实现 `__get__`、`__set__` 和 `__delete__` 特殊方法中的一个或多个，其实例作为另一个类（托管类）的类属性。描述符管理托管类中托管属性的存取和删除，数据通常存储在托管实例中。

名称改写（**name mangling**）

Python 解释器在运行时自动把私有属性 `__x` 重命名为 `_MyClass__x`。

魔术方法（**magic method**）

同特殊方法。

奶酪店（**Cheese Shop**）

Python 包索引（Python Package Index，PyPI，https://pypi.python.org/pypi）原来的名称，以“巨蟒剧团”表演的幽默短剧《奶酪店》命名。虽然是奶酪店，但是店里却什么奶酪都没有。写作本书时，https://cheeseshop.python.org 这个别名链接还有效。参见 PyPI 词条。

内置函数（**built-in function，BIF**）

随 Python 解释器一起提供的函数，使用底层实现语言（也就是说，CPython 用 C 语言，Jython 用 Java，以此类推）编写。这个术语通常指代无需导入就能使用的函数，参见 Python 标准库参考手册中的“2. Built-

in Functions”一章（http://docs.python.org/library/functions.html）。不过，内置的模块（如 sys、math、re 等）也包含内置函数。

平坦序列（**flat sequence**）

这种序列类型存储的是元素的值本身，而不是其他对象的引用。内置的类型中， str、bytes、bytearray、memoryview 和 array.array 是平坦序列；而 list、tuple 和 collections.deque 是容器序列。参见容器词条。

浅复制（**shallow copy**）

一种对象副本，引用源对象的全部属性对象。请与深复制相比较。另见别名词条。

强引用（**strong reference**）

让对象始终存在于 Python 中的引用。请与弱引用相比较。

切片（**slicing**）

使用切片表示法生成序列的子集，例如 my_sequence[2:6]。切片经常复制数据，生成新对象；然而，my_sequence[:] 是对整个序列的浅复制。不过，memoryview 对象的切片虽是一个 memoryview 新对象，但会与源对象共享数据。

容器（**container**）

包含其他对象引用的对象。Python 中的大多数集合类型都是容器，不过有些不是。请与平坦序列相比较，这种序列是集合，但不是容器。

弱引用（**weak reference**）

一种特殊的对象引用方式，不计入指示对象的引用计数。弱引用使用 weakref 模块里的某个函数和数据结构创建。

上下文管理器（**context manager**）

实现了 `__enter__` 和 `__exit__` 特殊方法的对象，在 `with` 块中使用。

蛇底式（**snake_case**）

标识符的一种命名约定，使用下划线（`_`）连接单词，例如 `run_until_complete`。PEP-8 把这种风格称为“使用下划线分隔的小写单词”，建议用于命名函数、方法、参数和变量。PEP-8 建议包名直接把各个单词拼接起来，不使用分隔符。Python 标准库中有很多使用蛇底式命名的标识符，不过也有单词之间没有分隔的标识符（例如，`getattr`、`classmethod`、`isinstance`、`str.endswith`，等等）。参见驼峰式词条。

深复制（**deep copy**）

复制对象时把对象的所有属性一起复制。请与浅复制相比较。

生成器（**generator**）

使用生成器函数或生成器表达式构建的迭代器，无需迭代集合就可能生成值。生成斐波纳契数列的生成器是个典型示例，这是一种无穷数列，在集合中绝对放不下。这个术语除了表示调用生成器函数得到的对象之外，有时还表示生成器函数。

生成器表达式（**generator expression**）

放在括号里的表达式，句法与列表推导一样，不过返回的不是列表，而是生成器。生成器表达式可以理解为列表推导的惰性版本。参见惰性求值词条。

生成器函数（**generator function**）

定义体中有 `yield` 关键字的函数。调用生成器函数得到的是生成器。

实参（**argument**）

调用函数时传给函数的表达式。按照 Python 习惯的说法，实参和

形参几乎等价。关于二者的区别以及各自的用途，参见形参词条。

视图（**view**）

在 Python 3 中，视图是一种特殊的数据结构，由字典的 `.keys()`、`.values()` 和 `.items()` 方法返回，作用是在不重复数据的前提下，提供字典的键和值的动态视图。在 Python 2 中，那些方法返回的是列表。字典视图都是可迭代的对象，支持 `in` 运算符。此外，如果视图引用的元素都是可散列的对象，那么视图还实现了 `collections.abc.Set` 接口。`.keys()` 方法返回的视图都是这样；对 `.items()` 方法返回的视图来说，如果其中的值都是可散列的对象，那么也是如此。

视为有害（**considered harmful**）

Edsger Dijkstra 写过一封题为“Go To Statement Considered Harmful”的信函，这为批评计算机科学技术的文章提供了一种标题格式。维基百科中的“Considered harmful”一文（http://en.wikipedia.org/wiki/Considered_harmful）列出了很多这种文章，包括 Eric A. Meyer 写的“Considered Harmful Essays Considered Harmful”（http://meyerweb.com/eric/comment/chech.html）。

属性（**attribute**）

在 Python 中，方法和数据属性（即 Java 术语中的“字段”）都是属性。方法也是属性，只不过恰好是可调用的对象（通常是函数，但也不一定）。

特殊方法（**special method**）

名称特殊的方法，首尾各有两条下划线，例如 `__getitem__`。Python 中的特殊方法几乎都在 Python 语言参考手册中的“3. Data model”一章（https://docs.python.org/3/reference/datamodel.html）做了说明，不过在特定上下文中使用的个别特殊方法在文档的其他部分里说明。例如，映射的 `__missing__` 方法在 Python 标准库文档的“4.10. Mapping Types”一节（https://docs.python.org/3/library/stdtypes.html#mapping-types-dict）提到。

统一访问原则（**uniform access principle**）

Eiffel 语言之父 Bertrand Meyer 写道："不管服务是由存储还是计算实现的，一个模块提供的所有服务都应该通过统一的方式使用。"在 Python 中，可以使用特性和描述符实现统一访问原则。由于没有 `new` 运算符，函数调用和对象实例化看起来相似，这也体现了这一原则：调用方无需知道被调用的对象是类、函数，还是其他可调用的对象。

托管类（**managed class**）

使用描述符对象管理类中某个属性的类。参见描述符词条。

托管实例（**managed instance**）

托管类的实例。参见托管属性和描述符词条。

托管属性（**managed attribute**）

由描述符对象管理的公开属性。虽然托管属性在托管类中定义，但是作用相当于实例属性（即各个实例通常有各自的值，存储在储存属性中）。参见描述符词条。

驼峰式（**CamelCase**）

标识符的一种命名约定，单词的首字母大写，然后连接起来（例如 `Connection RefusedError`）。PEP-8 建议类名使用驼峰式，但是 Python 标准库没有遵守这个建议。参见蛇底式词条。

文档字符串（**docstring**）

documentation string 的简称。如果模块、类或函数的第一个语句是字符串字面量，那个字符串会当作所在对象的文档字符串，解释器把那个字符串存储在对象的 `__doc__` 属性中。另见 doctest 词条。

瑕疵（**wart**）

指 Python 语言的不足。Andrew Kuchling 发表过一篇著名的文章——"Python warts"，仁慈的独裁者承认，他在设计 Python 3 的过程中受

此文影响，决定不向后兼容，否则无法修正大多数缺陷。Kuchling 提到的多数问题在 Python 3 中修正了。

像文件的对象（**file-like object**）

官方文档使用的一个非正式称呼，指代实现了文件协议的对象，有 `read`、`write` 和 `close` 等方法。常见的变体有：逐行读写，包含编码字符串的纯文本文件；作为保存在内存中的纯文本文件的 `StringIO` 实例；包含未编码的字节的二进制文件。最后一种可能有缓冲，也可能没有缓冲。从 Python 2.6 起，这些标准文件类型的抽象基类在 `io` 模块里。

像字节的对象（**bytes-like object**）

泛指字节序列。最常见的像字节的类型有 `bytes`、`bytearray` 和 `memoryview`；不过，支持低层 CPython 缓冲协议的对象，如果元素是单个字节，那么也属于此类。

协程（**coroutine**）

用于并发编程的生成器，从调度程序，或者通过 `coro.send(value)` 方法从事件循环中接收值。这个术语可以表示通过调用生成器函数获得的生成器函数或生成器对象。参见生成器词条。

形参（**parameter**）

声明函数时指定的零个或多个“形式参数”，这些是未绑定的局部变量。调用函数时，传入的实参（“实际参数”）会绑定给这些变量。在本书中，我尽量使用实参指代传给函数的实际参数，使用形参指代声明函数时使用的形式参数。然而，并不一定会始终这样做，因为 Python 文档和 API 经常混用形参和实参。参见实参词条。

虚拟子类（**virtual subclass**）

不继承自超类，而是使用 `TheSuperClass.register(TheSubClass)` 注册的类。参见 `abc.ABCMeta.register` 方法的文档（https://docs.python.org/3/library/abc.html#abc.ABCMeta.register）。

序列（**sequence**）

泛指长度（例如，`len(s)`）固定，可以使用从零开始的整数索引（例如 `s[0]`）获取元素的数据结构。Python 出现伊始，序列这个词就存在了，不过直到 Python 2.6 才由 `collections.abc.Sequence` 确定为一个抽象类。

序列化（**serialization**）

把对象在内存中的结构转换成便于存储或传输的二进制或文本格式，而且以后可以在同一个系统或不同的系统中重建对象的副本。`pickle` 模块能把任何 Python 对象序列化成二进制格式。

鸭子类型（**duck typing**）

多态的一种形式，在这种形式中，不管对象属于哪个类，也不管声明的具体接口是什么，只要对象实现了相应的方法，函数就可以在对象上执行操作。

一等函数（**first-class function**）

在语言中属于一等对象的函数（即能在运行时创建，赋值给变量，当作参数传入，以及作为另一个函数的返回值）。Python 中的函数都是一等函数。

引用计数（**refcount**）

CPython 内部对各个对象的引用计数，用于确定垃圾回收程序何时销毁对象。

用户定义的（**user-defined**）

在 Python 文档中，用户这个词几乎都是指我和你，即使用 Python 语言的程序员。用户与实现 Python 解释器的开发者是相对的。因此，“用户定义的类”表示使用 Python 编写的类，而不是使用 C 语言编写的内置类，如 `str`。

预激（**prime**，动词）

在协程上调用 next(coro)，让协程向前运行到第一个 yield 表达式，准备好从后续的 coro.send(value) 调用中接收值。

元编程（**metaprogramming**）

编写的程序使用程序的运行时信息改变程序的行为。例如，ORM 可能会内省模型类的声明，确定如何验证数据库记录里的字段，以及如何把数据库类型转换成 Python 类型。

元类（**metaclass**）

实例为类的类。默认情况下，Python 中的类是 type 类的实例；例如，type(int) 得到的结果是 type 类，因此 type 是元类。用户可以通过扩展 type 类定义元类。

元组拆包（**tuple unpacking**）

把可迭代对象中的元素赋值给多个变量（例如，first, second, third == my_list）。Python 高手通常使用这个术语，不过也有人使用可迭代对象的拆包。

真值（**truthy**）

只要 bool(x) 返回 True，x 就是真值。需要布尔值时，Python 会隐式使用 bool 计算对象，例如控制 if 和 while 循环的表达式。与此相对的是假值。

指示对象（**referent**）

引用的目标对象。谈及弱引用时最常使用这个术语。

装饰器（**decorator**）

一个可调用的对象 A，返回另一个可调用的对象 B，在可调用的对象 C 的定义体之前使用句法 @A 调用。Python 解释器读取这样的代码时，会调用 A(C)，把返回的 B 绑定给之前赋予 C 的变量，也就是把 C 的定义体换成 B。如果目标可调用对象 C 是函数，那么 A 是函数装饰器；如果 C 是类，那么 A 是类装饰器。

字节字符串（**byte string**）

可惜，在 Python 3 中仍然使用这个名称指代 `bytes` 或 `bytearray`。在 Python 2 中，`str` 类型其实是字节字符串，为了把 `str` 和 `unicode` 字符串区分开，才用了这个名称。在 Python 3 中没理由继续使用这个术语了，泛指字节序列时，我都尽量使用字节序列（byte sequence）这个术语。

作者简介

Luciano Ramalho 在 1995 年 Netscape 首次公开募股以前就是一名 Web 开发者了，他先后用过 Perl 和 Java，1998 年开始使用 Python。自那以后，他在巴西的几个新闻门户网站工作，使用 Python 做开发，还为巴西的媒体、银行和政府部门做 Python Web 开发培训。他经常在开发者大会上演讲，比如 PyCon US（2013）、OSCON（2002、2013 和 2014），还有多年在 PythonBrasil（在巴西举办的 PyCon）以及 FISL（南半球最大的 FLOSS 大会）上做过的 15 次演讲。Ramalho 是 Python 软件基金会的成员，还是巴西第一个众创空间 Garoa Hacker Clube 的联合创始人。他也是培训公司 Python.pro.br 的共同所有人。

关于封面

本书封面的动物是纳马沙蜥（学名：Pedioplanis namaquensis），身体细长，有一条呈红棕色的长尾巴。这种沙蜥身体为黑色，有四条白纹；四肢呈棕色，带白点；腹部为白色。

纳马沙蜥白天活动，是速度最快的蜥蜴之一。它们栖息在草木稀疏的沙砾平地，冬季在灌木丛边挖的洞穴里休眠。纳马沙蜥分布于纳米比亚全境的干旱稀树草原和半荒漠地区，以小昆虫为食。在 11 月，雌性会产下 3~5 枚蛋。

O'Reilly 出版的图书，封面上很多动物都濒临灭绝。这些动物都是地球的至宝。如果你想知道如何保护这些动物，请访问 animals.oreilly.com。

封面图片出自 Wood 的 *Natural History, Vol 3*。

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

- 微博 @图灵教育 : 好书、活动每日播报
- 微博 @图灵社区 : 电子书和好文章的消息
- 微博 @图灵新知 : 图灵教育的科普小组
- 微信 图灵访谈 : ituring_interview，讲述码农精彩人生
- 微信 图灵教育 : turingbooks
